THE ANDREW SCHULMAN PROGRAMMING SERIES

DOS INTERNALS

DISK
Included

INCLUDES MEMORY AND DISK
EXPLORATION UTILITIES

GEOFF CHAPPELL

DOS Internals

Geoff Chappell

Series Editor
Andrew Schulman

Addison-Wesley Publishing Company
Reading, Massachusetts Menlo Park, California New York
Don Mills, Ontario Wokingham, England Amsterdam Bonn
Sydney Singapore Tokyo Madrid San Juan
Paris Seoul Milan Mexico City Taipei

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademark. Where those designations appear in this book and Addison-Wesley was aware of the trademark claim, the designations have been printed in initial capital letters.

The author and publishers have taken care in preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

Library of Congress Cataloging-in-Publication Data
Chappell, Geoff.
DOS internals / by Geoff Chappell.
p. cm.
Includes index.
ISBN 0-201-60835-9
1. MS-DOS (Computer file) 2. PC-DOS (Computer file) I. Title.
QA76.76O63C452 1994
005.4'469--dc20 93-46019
CIP

Copyright © 1994 by Geoff Chappell

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. Printed in the United States of America. Published simultaneously in Canada.

Managing Editor: Amorette Pedersen
Set in 10.5-point Galliard by Benchmark Productions, Inc.

123456789-MA-9897969594
First printing, January 1994

Addison-Wesley books are available for bulk purchases by corporations, institutions, and other organizations. For more information please contact the Corporate, Government, and Special Sales Department at (617) 944-3700 x2915.

To A.,
You can never be alone
while you can laugh at Buñuel

Contents

Part I—Memory Management 1

Memory Problems 1

DOS 5 Solutions 3

Chapter 1—System Configuration 7

Automatic Configuration 7

User-level Configuration 8

Configuring IO.SYS 8

Overriding the BIOS 8

Reverting to Old Behavior 9

Configurable System Support 11

Configuring the Kernel 12

Installable Device Drivers 16

Running Programs 21

Four DOS Regimes 22

Chapter 2—The System Footprint 27

IO.SYS 27

The IO.SYS Loader 29

Low and High Segments 31

Int 19h—Preparing for a Warm Boot 33

Int 15h—Preparing for Ctrl-Alt-Del 48

Int 13h—Disk Support 54

Int 2Fh—Multiplex Interfaces to IO.SYS 57

Applications for Int 2Fh Function 13h 59

Windows Support 96

Error Conditions in Critical Sections 96

Instance Data 97

Measuring DOS's HMA Usage 102

MSDOS.SYS—The DOS Kernel 104

Low and High Segments Revisited 105

ROM-Executable DOS 106

Kernel Variations 110

The True DOS Version 112

Kernel Variables 113

Windows Support 115

Windows 3.0 Real Mode 122

Windows/386 123

Driving Windows from DOS 124

The Command Processor 127

Chapter 3—The Startup Sequence 131

Juggling the System 131

Before DOS 5 131

DOS 5 132

Activating the DOS Kernel 135

Moving Kernel Data to Low Memory 136

Planning Ahead for Moving Kernel Code 136

Marking Progress 137

Calling the Kernel 137

Initializing the Default Device Drivers 138

Interrupt Vectors for DOS Services 139

First Steps to Memory and Process Management 140

Booting DOS Externally 140

The SYSVARS Structure 142
Enabling the File System 143
Processing CONFIG.SYS 145
The Parsing Buffer 145
Configuration Block Analysis 151
Multiple Passes 154
Stabilizing the System 155
Loading DoubleSpace Early—How Is It Done? 156
Modified Device Driver Initialization 157
Moving DBLSPACE.BIN Away 159
Setting Up Drives for DoubleSpace 160

Chapter 4—Memory Allocation 163
Allocation Strategy and Upper Memory Linkage 163
A Practical Example—Selecting Memory Blocks by Address 168
Walking the Memory Arena Chain 183

Chapter 5—Program Execution 193
The Program Environment 196
Executable File Formats 197
Preparing for DOS Exceptions 199
Memory Layout for .EXE Programs 201
Relocation for .EXE Programs and Overlays 203
Memory Layout for .COM Programs 205
The Execution State 205
SPOTLOAD 209
Changing FILES= After CONFIG.SYS 226

Part 2—Some Development Tools 241
A Sample—Demonstrating the Memory Fudge for Devicehigh 241

Sample Two—Loading an Arbitrary Program as a Device Driver 243

Resident Programs 246

Chapter 6—Resident Programming in C **249**

Segmentation 250

Rearrangement 252

Swapping Segments 258

Swapping References 262

Debugging 270

Support 273

  Releasing System Resources 273

Terminating and Staying Resident 275

Interrupt Control 285

Avoiding the Compiler's Library Calls 287

Building the Libraries 289

An Example—Importing Material for Residence 292

Another Example—Writing a Resident Program 295

Chapter 7—Writing Device Drivers as C Programs **301**

The Essential Components 301

  C Functions for Driver Commands 303

  Unknown Commands 304

  A Front End 306

Initialization in C 311

  Using DOS Functions 311

  Using a Run-time Library 312

  Transparency 313

  Help From DOS 5 315

  Hindrance from Earlier Versions 316

The Run-time Interface Library 316

An Example—Loading a Resident Program as a Device Driver 331

An Example—A Real Device Driver 332

Extended Memory Access 332

File Emulation 333

Initializing the Device 340

Managing the Extended Memory Block 340

The Completed Driver 344

Operating Modes for Devices 345

Part 3—Extended Memory Management 349

The Interface In Outline 349

Chapter 8—Extended Memory Access from Real Mode 355

The Descriptor Cache 356

The 80386 Flat Memory Model 357

Finding the Segment Limit 357

Changing the Segment Limit 362

Virtual Machine Detection 369

CPU Identification 370

Instability of Flat Memory in Real Mode 372

The 80286 LOADALL Instruction 373

The 80386 LOADALL Instruction 379

Chapter 9—The A20 Line 387

Local, Global, and Physical 388

HIMEM's A20 Handlers 399

External A20 Handlers 402

Chapter 10—Extended Memory Sources 413

Int 15h 413

VDISK 414

Cooperation 416

Noncontiguous Extended Memory 418

Shadow RAM 419

Memory Above 64MB 420

Chapter 11—Disbursing Extended Memory 423

The HMA 423

In-line XMS Calls from C 425

Extended Memory Blocks 427

DOS 6 HIMEM Facility for Enumerating EMBs 438

Chapter 12—HIMEM.SYS at Close Range 441

Command Line Configuration 441

XMS Implementation 442

On the Unacceptability of HIMEM 3.xx and the Reallocate EMB Function in General 456

Chapter 13—Using the XMS 459

An XMS Clipboard 459

Task Switching 466

Initialization 472

Memory from SMARTDrive 478

The SMARTDRV.SYS Interface 479

SMARTDRV.EXE (SMARTDrive 4.0) Interface 482

DOS Reentrancy 488

The DOS Kernel's Sidedoor Interface 489

The Completed XMSFILE 503

Chapter 14—Extending the XMS **505**

Hookability 505

Tracing the XMS Chain 510

A Front End for XMS Extension 513

Spying on XMS Handle Allocations 519

XMS Modification by Task Switchers for Windows Standard Mode and the DOS Shell 538

Chapter 15—An Extended Memory Monitor **551**

A Brief Tour of Different XMS Environments 557

Windows 3.0 Standard Mode 559

Configuring SMARTDRV.SYS 567

Fragmented Extended Memory and DOSSWAP 573

Demonstration of Serious Flaws with HIMEM's Reallocate EMB 577

Windows Enhanced Mode 582

Part 4—Disk Management **585**

Chapter 16—Low-level Disk Support **587**

Absolute Disk Read and Write 587

Hard Disk Partitioning 591

The Partition Table 592

Primary DOS Partitions 593

Extended DOS Partitions 596

Drive Numbering 598

Boot Sectors 599

Hard Disk Boot Sectors 601

OEM Name and Version 601

Accepted Hard Disk Formats 602

Unrecognized Hard Disk Formats 603

Floppy Disk Drives 605

Floppy Disk Boot Sectors 606

Reliance on the Media Descriptor 607

The Default Block Device Driver 608

The IO.SYS Drive Tables 609

Int 2Fh Access to the IO.SYS Unit Tables 612

Finding the BIOS Drive Number 614

From Device Driver to File System 621

The DPB 625

Coordinating Directory Trees Across All Drives 625

Logical Drive Mapping 633

Asynchronous Drive-Swap Notification 636

Building a Logical Drive Map 638

Scanning Available Drives 641

Presenting Drive Descriptions 645

Generic IOCTL 654

Querying Support for Specific Minor Codes 656

Device Parameters 659

Disk Access Track by Track 661

Media Identification 663

Get/Set Access Flag 665

Media Type 665

Chapter 17—Error Codes 667

Error Code Translation 667

Standard Error Messages 668

Error Code Translation by COMMAND.COM 670

Parse Errors 675

Critical Errors 676
Network Errors 677
Extended Error Information 680
Retrieving Extended Error Information 682
Preserving Extended Error Information—a Note for TSRs 684

Chapter 18—Duplicating Disks via Extended Memory 687
Disk Identification 695
Disk Access 700
Temporary Storage 707
Conventional Memory 712
Extended Memory 714
The Complete Program 717
DoubleSpace 718

Index 721

Preface

Scope

It should be understood from the outset that this book is an analysis primarily of DOS 5 and 6. Constraints on time (and, let it be said, interest) have not always permitted special cases arising from earlier DOS versions to be taken up for discussion.

There are programming examples in this book which insist on having DOS 5 or higher. The most notable case is a library (developed in Chapter 7) which binds device driver initialization to the C run-time library. The code avails itself of the safety feature new to DOS 5 by which device drivers are supplied with an upper bound on the memory they may use during initialization. No doubt some vaguely satisfactory workaround could be contrived to support earlier DOS versions, but the attempt is not made. The decision in this particular case is borderline but my attitude is to not support DOS versions before 5 if the effort would be out of proportion to (and therefore distract from) the didactic point being made at the time.

Going Back

That said, you will find that analysis usually extends backwards to versions 4.01 and 3.30, both of which I have studied in comparable detail. Here again, there are decision boundaries about how much effort to spend over, say, the description of an interface that has changed since one of these versions. An example (from Chapter 16) is the undocumented int 2Fh function 08h through which IO.SYS supports the definition by DRIVER.SYS of extra units for the default block device driver. The structure in which IO.SYS maintains all its knowledge of block device units changed markedly for DOS 4 and, as far as this book is concerned, it must suffice just to note that the structure given in the text applies only from DOS 4 onward. I hope that all differences of any significance have at least been noted in this fashion.

Earlier versions than 3.30 were already uncommon by the time I wrote my first DOS program in 1989. Viewed through my eyes, all versions before 3.30 belong to some dim pre-history on which weak light is thrown occasionally by inspection of software that supports a range of DOS versions. A particularly well-informed source of information about earlier versions (and which has sufficient importance to justify close inspection) is Microsoft Windows, which supports DOS versions from 3.10 onwards.

As far as this book is concerned, DOS 3.10 is an event horizon. Put another way, to anyone concerned with programming DOS at system-level, DOS versions 1.xx, 2.xx, and 3.00 may as well lie in a black hole.

It is as well to offer system-level programmers some guidance over DOS versions while the boundaries and prejudices are being explained. DOS 1.xx appears to have been regarded generally as off-limits for many years: it is well-known that this version has no notion of directories or file handles, let alone support for memory management and program execution. These few words in this paragraph are all that will be said about DOS 1.xx in this book. No account is even allowed for DOS 1.xx in the programming examples, all of which assume that the case will be handled by the C run-time. I cannot resist the facetious remark however, that DOS programmers who cut their work off from the possibility of running under DOS 1.xx are doing themselves out of some small slice of the market: I know at least one person who continues to use (for normal secretarial work, no less) a genuine IBM PC running DOS 1.10 from 320K floppy disks.

For the most part, DOS 2.xx offers all the facilities that will usually be required by programs which work at the level of files and above, i.e., by ordinary, well-behaved applications. There are, however, good grounds for even applications programmers to insist on some form of DOS 3, at least. It may seem a bit excessive to discount a whole major DOS version for the following little reason, but this is what I recommend: In DOS 2.xx, there is apparently no way to fail an int 24h exception. True, it is possible to fiddle with registers on the stack and return to the application rather than to the DOS kernel, but Chapter 5 shows that this technique is suspect even under later DOS versions, primarily because of the difficulty of determining which DOS function has encountered the error, and thence what error indication (if any) should be prepared for the AX and flags registers. This lack of a Fail option complicates critical error handling unreasonably. Supporting DOS 2.xx will often involve implementing a full int 24h handler to be sure that if the user wants to terminate the program, then at least the cleaning up will remain within the program's control. Even with this extra work, however, the quality of error recovery cannot hope to be as good as can be achieved in later versions by the simple expedient of just failing DOS functions that encounter critical errors.

When it comes to systems-level work, you don't have to be at all adventurous to run into a plethora of old wives' tales. It is unimaginable that anyone competent will ever track down all OEM variants of DOS 2 and analyze them to settle some of these hoary old stories. No doubt the cases which were found important for TSR programming at the time are well established, but they are unlikely to have been covered exhaustively. This is why I describe them as old wives' tales: They will be rough and ready formulations that seemed to get the job done. Handwaving and folklore have no place in this book; at least, that's the intention. The spirit of this book is that programs should be safe and that programming should be sure. Knowledge brings confidence that ambitious projects can be made to work reliably, preferably first time.

To some extent, the question whether to support DOS 2.xx is academic, since I contend that even DOS 3.00 is not a sufficiently reliable platform for adventurous DOS programs. The most glaring difficulty for TSR programming is that there is no means to restore extended error information. It looks as if DOS 3.00 was a trial run preceding proper support for networking. Internal structures and functionality seem not to have been finalized and, perhaps inevitably, there are omissions.

In Microsoft's scheme of networking, a remote machine's request for file I/O is treated very much like the same request from a pop-up TSR. The desire to support networking appears to have concentrated the mind wonderfully on questions of re-entrancy and the nature of certain system resources. A consequence of good planning for network support is that the DOS kernel has enjoyed remarkable stability from DOS 3.10 onward.

Windows requires DOS 3.10 or higher. This is surely no coincidence. The questions noted above are also the ones of main interest when programming TSRs or more generally any sort of task-switcher. For a large class of programming exercises that lie beyond ordinary well-behaved access at file level, the general self-consistency and uniformity of the DOS programming model in versions 3.10 and higher, coupled with the justification that this version is required by the ubiquitous Windows, suggests that is wholly reasonable for systems-level programmers to insist on having at least DOS 3.10 present when their programs run.

It happens that concerns related to task-switching are not the whole story and that DOS 3.10 might not suffice for some applications. Although the interesting questions of process and resource management seem to have been thought out respectably well for 3.10, it was not until DOS 3.20 that attention was directed to giving DOS applications the means to exercise fine control over disk structure, that is, to call through to device drivers for specialized services like disk formatting. Generic IOCTL (used for the disk duplicating program presented in Chapter 18) is available only from DOS 3.20 onward.

Well, all new DOS versions bring new functionality that applications may use in their dealings with the system (or so one might have supposed until recently) and there will always be reasons to restrict some program's operational domain to some narrow range of probably recent DOS versions. These notes are an attempt to explain my particular feelings on the matter. In fact, close inspection of the programming examples accompanying this book will show that checking the DOS version is the least priority.

Looking Ahead

Some readers will wonder about DOS 6. This is unfortunate, for I think it shows that even those sections of the trade press which concern themselves with technical aspects of DOS programming have given DOS 6 rather more attention than it deserves.

Let there be no mistake: The programming model offered by DOS 6 differs hardly at all from the one provided by DOS 5. The resident portion of IO.SYS and the whole DOS kernel (MSDOS.SYS) are barely changed. As some measure of this, the *most* significant point I can see to make about the DOS 6 kernel is that it fixes a bug by which the DOS 5 kernel, when configured for **dos=high** with BUFFERS in the HMA, would lose track of error conditions when reading sectors into the BUFFERS system. The possibility has not yet been discounted that the simple oversight responsible for this had already been corrected in later releases of DOS 5.

A graphic illustration of the similarity between DOS 5 and DOS 6 at system level is that a DOS 5 machine may be booted in perfect safety, gaining the facility for multiple configurations, if the DOS 5 IO.SYS is updated to the DOS 6 version *with everything else being left intact*. Not even for DOS versions 3.20 and 3.30 is it possible to boot with IO.SYS from one version and MSDOS.SYS from the other.

Fortunately, there is more to DOS 6 than the system files. Quite a number of substantial applications have been bundled with DOS 6 and these presumably justify incrementing the major version number.

It is the book's overall policy to deal only with DOS features that are somehow intrinsic to the DOS programming model and that will be encountered by ambitious DOS programmers. This serves to excuse no end of analysis of Windows, sometimes reaching far beyond what some may think appropriate in a book on DOS. If the presence of some piece of software may cause the DOS programmer some difficulty and is a matter of sufficient interest to have held my attention for long enough to write about it, then it may very likely appear in the book.

Thus, it is one thing to examine HIMEM.SYS as the standard XMS server and to consider how its behavior is driven by the installation of other software such as the task-switchers for the MS-DOS Shell and Windows 3.1 standard mode. Throughout DOS's twilight years (and it could be a surprisingly long twilight), the XMS seems likely to be an important feature of DOS programming. It would be quite another thing however, were this book to study RAMDRIVE.SYS just because it happens to be an XMS client.

Similar arguments apply to DoubleSpace. I am aware that this disk compression software bundled with DOS 6 has generated enormous interest and that there is sore need for sober analysis. While Chapter 3 describes the interface between IO.SYS and DBLSPACE.BIN is some detail, this book is not the place for a discussion of DoubleSpace itself. Though DBLSPACE.BIN is billed as a new system file, this role arises only as the means to secure its presence on bootable floppy disks, and that IO.SYS should know about the file in order to enable compressed drives as early as possible. In programming terms however, DBLSPACE.BIN is far from fundamental: By hooking into DOS's device driver communications, it presents the contents of a file (the DoubleSpace Compressed Volume File, or CVF) as a DOS drive. Thus, DBLSPACE uses the structures and interfaces DOS already supports for drive management. These you will find discussed in Chapter 16, since they need to be understood by any DOS programmer intending to work close-in with standard disk drive storage on PCs. As far as this is concerned, the only special consequence of DoubleSpace's distribution as a component of the standard DOS package is that users might reasonably have expected rather more in the way of safety and compatibility. That Microsoft can have fallen short in these factors when releasing the version supplied with the DOS 6.20 upgrade must count to the industry's undying shame.

Notations and Conventions

Inevitably, in a book whose programs are mostly written in C but whose author is arguably *not* a C programmer, a few things should be explained from the outset.

All C source for this book is intended for compilation under Microsoft C 6.0. Perhaps it need not be mentioned here (since the realization will follow from just a cursory glance at the source code), but the compiler is used very much as a tool for high-level assembly language. For the few modules written in assembly language, the expected assembler is MASM 6.0.

There will undoubtedly be some notational changes required for porting the C source code to one of the competing C compilers for DOS, but I expect this process to be fairly mechanical. Some important cases exist, however, where the code depends strongly on labels that are probably peculiar to the Microsoft C run-time library. This applies to the two programs presented in Chapter 5, for reasons that are completely unrelated. In one, it is a simply irresistable convenience to reach deep within Microsoft's implementation of the `spawn` family of functions in order to replace their common core but to keep their ability to do such things as to follow the PATH environment variable when searching for a program. This gives the book's programming example the power of a real program rather than an instructive gadget. Nonetheless, it is the instruction which is the program's claim to existence and readers should be prepared to do their own possibly heavy adaptation if they want to port the program to another compiler. It is hoped that the dependencies will be flagged well enough that the task should not be odious.

This point is strongest in Chapter 6, which develops a scheme for resident programming in C. The object is not to develop a universal scheme but to see how problems posed by the compiler at hand may be resolved through combining an understanding of DOS's behavior with the will to overcome difficulties. For a different compiler, there may be different problems, but again, all I can hope is to have marked out the course sufficiently clearly.

Standard Macros and Symbols

Among this book's header files is one which contains symbols and so forth, that the author prefers to have handy when programming in C:

```
/* standard.h */

#ifndef        _STANDARD_INC
#define        _STANDARD_INC

/* Logical operators and values */

#define        AND       &&
#define        NOT       !
```

```
#define     OR      ||

#define     FALSE   0
#define     TRUE    1               // for consistency with TRUE = NOT FALSE

#define     OFF     0
#define     ON      1

#define     CLEAR   0
#define     SET     1

/* Convenient data types */

typedef unsigned char       BYTE;
typedef unsigned int        WORD;
typedef unsigned long       DWORD;

typedef signed char         SBYTE;
typedef signed int SWORD;
typedef signed long         SDWORD;

typedef unsigned char       CHAR;

typedef int                 BOOL;

/* Macro for generating a far pointer from segment and offset */

#define MK_FP(seg,off) (((_segment) (seg)) :> ((void _based (void) *) (off)))

/* Macros to decompose 16-bit and 32-bit objects into high and low
    components and to reconstitute them */

#define HIGH(x) ((BYTE) ((x) >> 8))
#define LOW(x)      ((BYTE) (x))

#define MK_WORD(high,low)  (((WORD) (high) << 8) | (low))

#define HIGHWORD(x) ((WORD) ((x) >> 16))
#define LOWWORD(x)  ((WORD) (x))

#define MK_DWORD(high,low) (((DWORD) (high) << 16) | (low))

/* Macros for directing the compiler to use current segment register
    values rather than generate relocatable references */
```

```
#define     CODESEG              _based (_segname ("_CODE"))
#define     CONSTSEG    _based (_segname ("_CONST"))
#define     DATASEG              _based (_segname ("_DATA"))
#define     STACKSEG    _based (_segname ("_STACK"))

/* Macro for NULL in case using STDLIB.H would be inappropriate */

#ifndef NULL
#define NULL ((void *) 0)
#endif

#endif
```

Standard Makefile Inclusions

Likewise, all the book's makefiles include the following standard provisions, if only to hide awkward things like ensuring that compiler options don't sneak in from environment variables.

```
# standard.mak

# This is an NMAKE include file for projects containing assembler and C.

# Pre-defined inference rules present some problem when compiling C source
# files with the /Fa option to generate assembly listings. Once the .ASM
# file exists, NMAKE assembles the old listing rather than compile the new
# C source even if the inference rule contains an explicit .C dependence.
# Fix this by reversing the order of .ASM and .C in a new .SUFFIXES list.

.SUFFIXES :
.SUFFIXES : .c .asm

# Utilities not known to NMAKE by default:

LIBRARIAN = lib
LINKER = link
MAPSYM = mapsym

# Favored switches for the assembler, compiler and linker:

ASWITCHES = /Cp
CSWITCHES = /Gs /Oas /W3 /Zp1
LSWITCHES = /noi

# Generate listings unless "LISTING=OFF" is given on the NMAKE command
# line or precedes the inclusion of this file.

!IFNDEF LISTING
LISTING = ON
!ENDIF
!IF "$(LISTING)" == "ON"
ALIST = /Fl /Sg
CLIST = /Fa
LLIST = /m
!ENDIF

AFLAGS = $(ASWITCHES) $(ALIST)
CFLAGS = $(CSWITCHES) $(CLIST)
LFLAGS = $(LSWITCHES) $(LLIST)

ASSEMBLE = $(AS) /c $(AFLAGS)
COMPILE = $(CC) /c $(CFLAGS)
LINK = $(LINKER) $(LFLAGS)

# Pseudo-targets:

ALL : CLEAR TARGETS

# The first pseudo-target is a possibly over-cautious insistence on
# clearing flags passed to the assembler, compiler or linker through
# environment variables.

CLEAR :
  set $(AS)=
  set $(CC)=
  set $(LINKER)=
```

Header Files

Heavy use is made of a collection of header files defining structures, etc., for key elements of DOS programming. It is hoped that the use of long identifiers makes it easy enough to follow source without having to reprint these many header files in the text of the book or even as an appendix, especially when many of the structures concerned are presented in tabular form anyway. All include files appear on the accompanying disk.

In the inclusions at the top of each C module, it is assumed that header files from this collection are available on a path defined in the INCLUDE environment variable. The filenames are therefore enclosed in angle-brackets. For clarity, standard header files and those specific to this book are presented in two separate blocks of inclusions.

Nomenclature

In general, structure elements and public data are named with a prefix to describe the data type, following the so-called "Hungarian" scheme presented in the Windows SDK. Given the author's clear preference for unsigned quantities, only some of the prefixes given in the Windows SDK are used here:

b	int (two bytes) taking Boolean values
c	char (single byte)
dw	dword (four bytes)
lp	far pointer
p	pointer (typically a 16-bit offset, given the predominance of small memory model)
w	word (two bytes)

All the book's source code, whether for C or assembly language, is case-dependent. If a structure element, function or public label has a compound name, then each element begins with an upper case letter for readability, as in SomeFunctionName or dwLongStructureElement. Macros and other equates are usually given in capitals only, using underscores if they seem required for readability, as in SOME_MACRO. Names for automatic data and many static variables often consist of lower case letters only: since their scope is severely restricted, their readability (if not also the name itself) is perhaps not so important.

Typeface Conventions

Some attempt is made at using different typefaces to emphasise the technical role of a word or phrase in the book's main text.

When DOS commands such as **fdisk /status** are used mid-sentence, they appear in bold. The same applies to configuration statements like **multitrack=off** or **RemMediaCD ChngTrack=on** (which are permissable in CONFIG.SYS and SYSTEM.INI respectively) and, less commonly, to keywords from programming languages.

Filenames and program names are given in upper case when they are not in the context of a command. Thus, text might speak of the WIN.COM program that starts Windows but of running the command **win ::** to see Windows crash.

Function names that might ordinarily appear in programming code retain the use of a fixed pitch typeface when extracted into the main text, as happens for instance when a point is made about the `_chain_intr` function in the Microsoft C library. Fixed pitch is also used when taking several lines to show the contents of a text file or to present the results of running a program.

Acknowledgments

It seems customary in books that delve into DOS or Windows programming to thank a list of software developers who have helped with information, including some who must remain nameless. This is not the case here, but since this is my first book, please bear with me while I thank those who have played key parts in bringing me to a study of DOS programming, in addition to thanking those with a closer connection to the book itself.

Although he may not remember it, Professor Jonathan Kaye of the Department of Linguistics at the School of Oriental and African Studies (SOAS) played an instrumental role in seeding my over-indulgent interest in programming way back in 1989.

Without the students living at the International Hall of Residence at the University of London, some of whom used to insist on misbehaving with a few communal PCs (and now just have a canny knack of doing strange things with their own PCs), I might never have found the motivation to uncover the truth behind DOS's inner workings, or at least to put my findings into practice. While this book was being written, many of the Hall's students and staff (especially its Warden, Donald Mann) found time for words of encouragement. They probably think their efforts were not even noticed, so gloomy was my outlook at some points in this book's history.

Major Reg Cross and his wife Mary Cross have been consistently concerned for my well-being. Reg suggested many years ago that I had the right sort of mind for computing. I'm not sure I yet agree with him or that I've taken up the subject in quite the way he had in mind, but I know he'll take it in his usual good humor if I name him as the one to blame for all this.

Neither Miles Dennis nor Martyn Lovell is primarily a PC programmer, but it was invaluable having them at hand in the first few months to read over rough work for the book and to assure me that my mental flotsam was surviving its translation process. My respect for their technical abilities is no faint praise, as readers of this book will quickly appreciate. Miles also has been a consistently concerned and supportive friend. And I'm not saying that just because I've had near to permanent loan of his nice fast modem.

Had Miles and Martyn not insisted, I might never have joined CIX, a system for conferencing and email which first put me in touch with professionals who were interested in technical aspects of DOS programming. Immediately, I was encouraged to make something of my DOS study, which just a little earlier I had thought to abandon. I should like to single out David Jewell, author of the forthcoming book *Polishing Windows*, who has been particularly helpful by way of encouragement and always ready to act as a common-sense sounding board (until he too succumbed to this degenerative condition called book-writing).

Dan O'Brien, also met through CIX, may not have known what he was starting when he gave me the opportunity to review the book *Undocumented DOS* for *.EXE* magazine in April, 1991. On the other hand, he may have had a pretty good inkling that the review would be critical. This book is very much Andrew Schulman's Revenge. It is a double-edged blade however; Andrew has surely had many "heart attacks" while wondering how to cope with his author's curious approach to writing a book. He has taken a lot of trouble to see that the results actually appear in print, going so far as to write the introductions and links that I simply didn't do.

Ralf Brown was kind enough to read my scrappy manuscript and make sense of it. This can't have been easy, as you the reader are about to discover, in spite of the changes and second-thoughts prompted by Ralf's analysis.

At Benchmark Productions and Addison-Wesley, there are a host of people I have never seen but whose work is responsible for transforming my manuscript into the book you have before you. Those I know about are Abby Cooper, Ann Lane, Amy Pedersen, Chris Platt, and Andrew Williams. I have at least talked to Chris Williams and Phil Sutherland. If you like this book, please write to Phil at Addison-Wesley: in doing this, you may help restore them from the depths of despair to which I've no doubt reduced them.

Professor John Taylor, Director of the Centre for Neural Networks at King's College, London has been very kind to keep in touch and keep me in mind despite the ever-lengthening time that this book has prevented me from engaging in useful research (not to mention writing up results).

Special personal thanks must be extended to long-standing friends Brad McCusker, Anne Tully, and Lorna Gibb, not for any particular connection with the book, but for consistently showing far more faith in me than can be reasonably justified and for just about managing to counter-balance my persistent sense of abject failure. My parents, Myles and Angela Chappell, in Bundaberg, Australia, have been truly supportive despite my great distance from home. Without their assistance at vital moments, this book would certainly have been aborted. Please recommend this book heartily to all and sundry so that I can visit my parents and take my friends out to dinner.

When I was just a teenager, a kind, gracious gentleman by the name of Fred Hendricksen took a large sum of money from his own pocket for me to buy books. Now I've written one, Uncle Fred.

Geoff Chappell
(geoffc@cix.compulink.co.uk)
London

PART I

Memory Management

DOS exists to manage a microcomputer's resources, uniting them behind a consistent and stable interface for the benefit of application programs. It provides a file system for organized access to peripheral devices, a framework in which programs may cooperate over memory use, and the means by which one program may run another.

Much that struck users as new to DOS 5 concerned program execution, particularly the MS-DOS Shell's ability to suspend a program, switch to another, and resume later, rather than be tied to a sequence in which each program that loads another must wait for its child to cede control. Observe however that the operating system itself does not produce this task-switching effect. Programs may be switched in mid-execution because they have been loaded by a special program that knows how to seize control from a child. A task-switching program is no more fundamental to DOS than any of the other utilities that constitute the familiar DOS commands. Its development required little, if any, alteration to DOS at operating system level—as attested to by the commercial availability for some years of sophisticated task switchers from vendors unconnected with Microsoft.

That DOS is now furnished with a standard task switcher is of course welcome and important. This fact carries serious implications for programming practice which may be taken up elsewhere. This book however directs its attention at features of DOS 5 and 6 whose effects pervade all levels of DOS's construction and use. First and foremost among these is the extension of memory management to open up new areas of memory for programs. It also makes DOS itself less obtrusive in conventional memory (of which, historically, there has never seemed enough).

Memory Problems

DOS is an operating system for computers based on the 80x86 family of processors, running in real mode with 1MB of addressable memory. Almost invariably, the computer in question is a PC or compatible, which reserves much of that memory for video RAM, a ROM BIOS, and ROM extensions that accompany additional hardware, leaving DOS with only whatever RAM has been installed in the first 640K.

Any given machine is unlikely to use all the reserved area for hardware support. A variety of schemes exist to find the unused space and fill it with RAM so that DOS programs need not be constrained by this 640K barrier. An early solution employed memory chips on special boards with hardware to map this extra memory into the processor's address space. With an 80286 processor, similar hardware can remap memory that would otherwise lie above the 1MB limit of 20-bit addressing. The 80386 processor may be stepped up to protected mode to run the DOS session as a virtual-8086 task with paged memory. Whatever the method, one common factor is that for almost all its history DOS itself has been oblivious to the possible usable RAM above 640K. Up to version 5, DOS certainly did not attempt to coordinate its use.

Actually, DOS's ignorance of upper memory had some benefits, for it enabled bank-switched memory to be implemented with minimal disruption to normal programming. In these systems, an Expanded Memory Manager sets aside a window (a page frame) through which its clients can access vast amounts of memory a little at a time. An Expanded Memory Manager allows the client to specify which page is being viewed through the window at a given moment. This extra memory can be obtained by the mechanisms described above but can also be simulated by disk storage. Of course, a special interface—the Expanded Memory Specification (EMS)—is needed to support the bank switching, with the disadvantage that it automatically excludes a large amount of preexisting software. It risks alienating an ultimately larger amount of software that might choose not to take the trouble over a solution whose life would surely be limited by processor improvements. The ideal reconciliation places the page frame above 640K, safely outside DOS's domain, so that nonparticipant software does not suffer a reduction of its available memory to support the new regime. This is why the EMS page frame typically resides at an address such as E0000h.

Given that RAM for upper memory and RAM to be switched in and out through a page frame can be retrieved from the same source, it is not surprising that, historically, bank-switching and the provision of upper memory have been closely linked. An early version of the Expanded Memory Specification (LIM EMS 3.2) was indifferent to the source of extra storage but insisted on having RAM available in upper memory for the page frame. In many cases, the EMS drivers which implemented this specification did in fact serve both purposes, using a single mechanism both to provide bank-switched memory and to fill the gaps between video RAM and the ROM BIOS.

Some went further, recognizing that RAM mapped into the Upper Memory Area between 640K and one megabyte, but not needed for EMS page frames, could be presented in a form mimicking a chain of DOS memory blocks, which could then be linked with DOS's original chain in conventional memory. By adding to the structures DOS already uses to subdivide its memory arena, these expanded memory managers subsume control of upper memory beneath the int 21h interface and give programs access to usable space in upper memory at no extra cost. Dedicated memory, such as that used for video RAM, the ROM BIOS, or for an EMS page frame, simply appears to have been allocated to the memory manager. The familiar DOS functions manage the entire 8086 address space.

This ideal of uniformity has its limitations on a number of counts. First, the highly desirable goal of loading resident programs into the typically small blocks of upper memory, in order to conserve a large block of conventional memory, requires some distinction between the two memory chains. DOS offers a range of memory allocation strategies, but unless a program's .EXE header specifies a limit to the program's initial memory requirement, DOS has little choice but to seek the largest possible allocation when loading the program into memory for execution. Aside from the chance situation that the largest available block is nonunique, the allocation strategy will be irrelevant; indeed, so few programs are linked with a maximum allocation setting that some writers have mistakenly thought that DOS (inconveniently) resets the allocation strategy to first-fit whenever it loads a program. Only a relatively small number of programs can be forced into the upper memory area merely by setting the allocation strategy to best-fit or last-fit. EXEHDR can change the maximum-allocation setting of an existing EXE file, but this is hardly convenient.

A second factor is the truly amazing quantity of PC software, professionally written or otherwise, that makes unsupportable assumptions about its operating environment, particularly about its location in memory. Amidst the lamentations of programmers who feel constrained by a ten-year-old operating system and the well-deserved plaudits delivered to those whose clever work continues to extend DOS's capabilities well beyond original expectations, it is not stated often enough that much of DOS's commercial success is due to the willingness of its designers and others to accommodate poor programming practice in successful applications. In this particular case, it became apparent early on that some programs would not run correctly if loaded into an Upper Memory Block (UMB)—so many in fact that the DOS 5 manual claims "it is not unusual for your system to lock up while you are moving programs to the upper memory area."

Whether or not this assessment is unduly pessimistic (bear in mind that **loadhigh win /r** is among the problem cases), the consequence is that not only must a distinction be kept between the chain in upper memory and the one in conventional memory, but that permanent linkage of the two may be unsafe.

DOS 5 Solutions

With the release of DOS 5, memory managers that support UMBs are offered a standard method for dealing with the problems just mentioned. DOS does not itself enter into the hardware-dependent world of providing RAM for upper memory except for users of machines with an 80386 processor or better. For these machines it bundles an expanded memory emulator EMM386.EXE that was originally distributed with Windows and which uses the 80386 paging mechanism to map memory from above the first megabyte. Users may supply another memory manager, as indeed they must for lesser computers. Provided the memory manager can communicate the location of its UMBs, a simple **dos=umb** statement in CONFIG.SYS will enable control at DOS function level (int 21h function 58h), both of memory allocation strategy and of the linkage between conventional and upper memory. Specific applications of this control are built into DOS: The internal command **loadhigh** directs DOS to load a given program into upper memory if possible, and CONFIG.SYS admits a **devicehigh** statement that acts similarly for device drivers (but which works very differently at close inspection). Developers may, of course, prefer to exercise finer control over upper memory usage. As always, some programs will make better use of the facility than others.

The interface through which DOS expects to discover that a memory manager provides UMBs is actually a small component of a larger system (XMS; see below) primarily for accessing extended memory. Although DOS is aware of some protected-mode software, it does not itself access extended memory apart from that tiny area that lies immediately above the 1MB boundary and that is addressable from real mode using segment 0FFFFh. This High Memory Area (HMA) acquires its special property because the processors that support protected mode use more than 20 address lines to locate objects in memory. On the 8086, with only 20 bits in which to hold a physical address, the segment:offset pairs FFFF:0010 and 0000:0000 identify the same location. This is true of real mode on the 80286 and higher processors only if hardware disables the A20 address line to force strict addressing compatibility with the 8086. This is indeed the arrangement that most BIOSes adopt as standard and that programs written for execution in real mode are entitled to assume. When the A20 is enabled however, very nearly 64K (from FFFF:0010 up to and including FFFF:FFFF) of extended memory can be reached from real mode.

Access to the HMA must not be attempted without first verifying that the A20 line is enabled and that the HMA is in fact addressable (see Chapter 9 on the A20 line). At any given moment, the HMA may effectively not exist, so unlike upper memory, the HMA cannot be brought under the umbrella of normal DOS memory management. Even if the HMA were permanently accessible, it could be accommodated only at the int 21h level. Paragraph boundaries in the HMA cannot be represented by a segment only (with an implied offset of 0000h), so the system of memory arena headers used below int 21h level for memory management in the first megabyte would break down.

The Extended Memory Specification (XMS) provides not only a scheme for coordinating use of extended memory in general, but also an interface for manipulating and testing the A20 and for allocating the HMA. The statement **dos=high** in CONFIG.SYS directs DOS to obtain the HMA for its own purposes. By using the HMA for large portions of the DOS system files and command processor, for the **buffers** system, for a piece of the device driver HIMEM.SYS, and for fonts prepared by DISPLAY.SYS, up to 64K of conventional memory that would have been used by DOS can be made available to application programs instead.

XMS support is supplied in the first instance by HIMEM.SYS, which implements all of the XMS except (and this point has inexplicably been missed by other writers on this subject) for two functions concerned with UMB allocation, which must be added by the memory manager (such as EMM386) that actually provides the UMBs. (To repeat: HIMEM.SYS does not provide UMBs or UMB management. A memory manager such as EMM386.EXE does.) HIMEM, or a closely-matched replacement, is required for A20 control, for HMA management, and for access to extended memory that is not addressable from real mode. The last of these is important not only for supporting real-mode programs such as RAM drives and disk caches that use extended memory for data storage, but also for coordinating memory access across the transition from DOS's real-mode environment to protected mode and back again. A growing number of DOS-extended programs run in protected mode but that still require contact with DOS.

To meet the demands of the future, DOS must satisfy many different interests. Its new machinery for memory management is highly configurable. Later chapters examine the tools with which DOS opens itself to new areas of memory and prepares the way for protected mode software. However, memory is a resource to which the system as a whole is very sensitive, and it is essential to have a clear understanding of the context in which these tools operate and in which the resource is handled. To a large extent, memory management depends on the layout chosen during system startup: DOS's initialization and the system files at its heart were restructured to provide the new features in DOS 5 and 6. This is the point from which this book begins its study.

CHAPTER 1

System Configuration

From a programming perspective, DOS's main substance resides in its two system files. One of these—MSDOS.SYS, hereafter referred to as the operating system kernel—provides services for application programs via the familiar int 21h interface. It is by calling system functions defined in this interface that applications gain access to a unified treatment of files, memory, and various other communal resources.

To lend the operating system services some measure of device independence, the kernel does not communicate directly with the computer's hardware (for example, the ROM BIOS functions). Instead, it forces all interaction through a fairly narrow channel in which special programs called device drivers match the hardware to the needs of the DOS file system. It falls to the other system file—IO.SYS, which is in fact the first to be loaded—to assess the standard hardware for the machine in question and prepare a set of default device drivers.

Automatic Configuration

Configurability is offered on several planes, beginning for all practical purposes with the adaptation of default device drivers to accommodate known variations in standard hardware. IO.SYS does this automatically during initialization, relying on the BIOS to describe hardware properties via interrupt services or flags in the BIOS data area. For instance, the BIOS sets a bit flag at 0040:0096 if it detects the presence of an enhanced keyboard; in this case IO.SYS configures its default CON device driver to retrieve tokens from the keyboard buffer using newer int 16h functions (such as function 10h rather than function 0). Likewise, IO.SYS prepares its default block device driver (including assignment of drive letters) according to the results of int 13h inquiries about disk drive characteristics and also according to information read from disk.

As well as providing device drivers for default hardware, IO.SYS also lends general system support. The best-known example is the installation of an int 13h handler which deals with complications in transferring data to or from memory regions that cross DMA page boundaries. By correcting for DMA errors at the lowest level, everything else running on the computer—applications, the DOS kernel, and even the default block device driver—is spared from ever having to think about the problem. Chapter 2 discusses this and other forms of general assistance in more detail as part of a deeper analysis of IO.SYS. For now observe that most forms of this support are configured automatically, usually to overcome some problem with a particular BIOS variation or at least to standardize behavior.

Before leaving the system's automatic adaptation to hardware, note that separation of the system into two files gives manufacturers of non-PC hardware the opportunity to distribute IO.SYS in a modified form. It even allows them to replace IO.SYS with something of their own design. In this scheme, MSDOS.SYS is generic; the DOS seen by applications remains essentially unchanged but the default device drivers and general system support are configured for a non-standard machine architecture. Increasingly of course, users expect to boot a stable DOS straight out of an ordinary-looking box, whoever manufactured their computer, so this replaceability of IO.SYS is becoming of historical interest only.

Among the lessons to be learned from history is that although DOS understands its memory arena to extend from the base of memory upwards in a single piece, it does not itself impose an upper limit. The so-called 640K barrier is a feature of the PC architecture that maps video RAM into the CPU address space at the physical address A0000h. It is merely a convenience to regard the segment A000h as the barrier between conventional and upper memory.

User-level Configuration

User involvement in the configuration process is enabled through statements in a specially-named text file CONFIG.SYS. It is not attempted until the kernel has been initialized and the file system activated. Four distinct levels may be identified.

Configuring IO.SYS

First, the default device drivers already set up by IO.SYS may require additional reconfiguration because of some factor not discernible from the BIOS or because information supplied by the BIOS is known to be inaccurate or unsuitable.

Overriding the BIOS

Correction of an unsuitable default configuration is illustrated by the command **switches /k**, which directs the default CON device driver to disregard any earlier indication that an enhanced keyboard is attached; instead, the driver must use older int 16h functions, presumably because the user intends to run programs that find new keyboard tokens confusing (a few such programs are listed in the APPNOTES.TXT file distributed with DOS 5).

Inaccuracy may arise if the BIOS's understanding of the computer's hardware is acquired from a CMOS device with options describing standard equipment only. Suppose, for instance, that a storage device supported by the int 13h disk services is fitted but is sufficiently unusual in that it cannot be matched correctly to an available CMOS option; in this case, it may be possible to select the inaccurate CMOS description so that

IO.SYS at least knows of the disk drive's existence, then use the **drivparm** command to reconfigure the default block device driver with the true characteristics (before it tries to access the storage device). Block device drivers installed at user level (see below) are also subject to the **drivparm** command, which is in fact a front-end to a deep and flexible I/O Control facility for block device drivers generally (see Part 4).

The same command might also be used simply to tell the default driver that a disk drive the BIOS thinks can report media changes will in fact never do so, perhaps because of damage. This illustrates how configurability has the important side-effect of assisting the system to recover from unexpected situations. As another example, consider an old BIOS that uses the byte at 0040:0096 for some purpose that later became incompatible with the interpretation as a keyboard flag, perhaps inducing IO.SYS to think that an enhanced keyboard is attached when in fact one is not; the keyboard switch noted above can be used to increase compatibility.

Adaptation of a default device driver to accommodate some feature that cannot be discerned from the BIOS must be viewed as exceptional, on the grounds that such nonstandard hardware likely warrants an entire device driver of its own. Cynics may care to observe that neither CONFIG.SYS command in this category is documented in the standard DOS manual.

Reverting to Old Behavior

Setting **multitrack=off** directs the default block device driver to decompose all read/write requests so that each int 13h call involves sectors from one track only. This is the normal practice for floppy disk drives, since the standard controller cannot select a different head or seek to another cylinder without being reprogrammed. In general, controllers for hard drives are not restricted like this. At some stage in DOS's (relatively) early history, the default block driver understood that decomposition is not needed when accessing fixed disks. With many such improvements, it is a good idea to leave open some way of reverting to the old behavior just in case.

A notorious problem with the system clock's midnight flag is handled by the command **switches /t**, which is new to DOS 5. Every 24 hours, the BIOS resets its count of timer ticks and marks the event by preparing a nonzero midnight flag that it will return when the clock count is next consulted. (The count is usually synchronized at startup to have zero correspond to midnight). Some BIOS implementations increment the flag with each passing midnight, but others (and perhaps most) simply set the flag to the same nonzero value no matter how many days have elapsed. Unfortunately, the BIOS does not indicate which method is followed. Starting with DOS 5, the default behavior is to take advantage of the more sophisticated BIOSes for which the flag is an accurate tally of midnights passed. This is compatible with BIOSes for which the flag is constrained to be zero or one, as long as the clock is sure to be checked at least once a day. In the alternative **switches /t** configuration (which is needed only for BIOSes capable of returning a nonzero flag other than 01h), the default clock device driver reverts to the previous practice in which a nonzero midnight flag, no matter what the value, is taken to mark the passing of just one day.

The DOS Clock, Idle Loops, COMMAND.COM, and DOSKEY

Since the question of time-keeping is taken up nowhere else in this book, it may as well be noted here that from DOS 3.30 onward, the DOS kernel checks the time once every 256 loops while waiting for a character in console mode. This is the same idle loop which forms the basis of int 21h function 08h (Read Keyboard without Echo). It is implied by several others of the so-called console functions that read from stdin, such as int 21h function 0Ah (Buffered Keyboard Input, used by COMMAND.COM to read command lines), and also by int 21h function 3Fh when reading from any handle that refers to a device being operated in console mode.

This idle loop is one of those during which DOS may issue the int 28h scheduling hook that many TSRs rely on for their reactivation. It is also the loop in which the kernel sends int 2Ah function 84h (an undocumented call) as part of its scheme for signaling critical sections to multi-tasking software that needs better performance than can be achieved with the usual, documented methods. Windows enhanced mode, for instance, monitors these critical section signals so that it may know when parts of the underlying system may not be reentered. Int 2Ah function 84h is complementary to these danger signals: It announces an ideal opportunity for a multi-tasker to step in. Thus, the Windows Virtual Machine Manager (VMM) inside WIN386.EXE converts the DOS kernel's int 2Ah function 84h to its own `Release_Time_Slice` service.

For thoroughness, note that DOS schedules one other facility while waiting for a character in this loop. Providing that stdin refers to a character device and that the PRINT utility has not used the undocumented int 21h function 64h to inhibit the following measure, the DOS kernel sends the relevant device driver a special variant of the nondestructive read command (it passes a flag in the request header's status word). The default CON device driver in IO.SYS and its ANSI.SYS replacement both recognize this command as a signal to alert a wait routine in the BIOS of the PC-Convertible (if the computer is actually that model).

Anyway, the result of adding an occasional time check to these scheduling hooks is to ensure more or less that the default clock device driver keeps track of passing days if the computer is left idle at any sort of DOS-level command prompt—most notably the familiar C:\> prompt generated by COMMAND.COM. This is arguably the most practical solution—given the goal of device independence and a desire to have a clock device that does not increase the sometimes substantial load placed on processing timer-tick interrupts.

Unfortunately, the usefulness of the provision has been undone to some extent by the introduction of the DOSKEY utility in DOS 5. This utility has the subtle effect of implementing the COMMAND.COM prompt through int 16h calls to the BIOS instead of an int 21h function 0Ah (Buffered Keyboard Input) call to DOS. While waiting for keystrokes to become available, DOSKEY generates its own idle loop during which it issues int 28h and int 2Fh function 1680h (a newly documented scheduler call which Windows interprets as a slightly more general substitute for int 2Ah function 84h). However, DOSKEY does not arrange for the default clock device driver to check the midnight flag. The result is that a computer left overnight at a COMMAND.COM prompt may lose a day if running DOS 5 or higher with DOSKEY.

This is the first (and simplest) indication in this book that some admittedly intricate aspects of software development are being driven by experiment when deductive analysis would surely be better. A moment's reflection suggests that if DOSKEY's int 2Fh function 4810h (Read Command Line) is to replace the editing services built into int 21h function 0Ah, then it should also reproduce the scheduling hooks implied by that DOS function. Failure to do this, especially to issue int 28h, would be all too obvious when testing DOSKEY in the presence of some TSRs; for good reasons the official reference suggests that idle programs issue int 28h themselves (and int 2Fh function 1680h for similar reasons). There is probably not much that can be done about reproducing the special measure for the PC-convertible, but, if only for consistency, the advice should be carried through to recommending that idle programs call DOS's Get Time function (int 21h function 2Ch) occasionally also.

Configurable System Support

General system support offered by IO.SYS is a low-key affair, designed to hide effects even from the IO.SYS device drivers let alone the user level. The one configurable element is the system's support for a pool of stacks on which to process hardware interrupts and the NMI, rather than expect all applications to provide sufficiently large stacks. By default on modern machines, IO.SYS sets aside nine stacks of 128 bytes each, but a different arrangement may be specified with the **stacks** command. Without the communal stacks, interrupt handlers drain whatever stack is in use at the moment of interruption. Of course, no single interrupt handler is likely to demand an excessive stack size, but multiple interruptions in quick succession can cause problems. As with several options at the next level, configurability allows the user to balance the benefits of a particular feature against its cost in memory.

The Stack Sharing Scheme

To see the effect of hardware interrupts on a program's stack, start a debugger like DEBUG or SYMDEB and enter commands such as

```
-a
1204:0100 mov bx,0100
1204:0103 xor cx,cx
1204:0105 loop 0105
1204:0107 dec bx
1204:0108 jnz 0103
1204:010A int 3
1204:010B
-r sp 0200
-g
```

After letting execution run to the int 3 breakpoint, inspect the stack (say, the 40h bytes leading up to offset 0200h). The amount of stack space used while the CPU idles in so many loops is easily seen because the memory concerned has been cleared beforehand. In the unlikely absence of hardware interrupts, only the three words anticipated for the INT 3 instruction will be pushed onto the stack. Applications should be aware, however, that their stack space is used without their explicit direction—as should be clear enough in the above demonstration, since hardware interrupts are almost certain to occur.

Readers are encouraged to experiment with this simple test rig under different system configurations. It is observed later that the stack pool is constructed only after all device drivers have been loaded. This means, for instance, that the usefulness of the stack-sharing scheme is reduced by the presence of KEYB (which replaces the int 09h handler, effectively removing keyboard interrupts from the scheme) and by the choice of MOUSE.COM over MOUSE.SYS. In configurations that pass all hardware interrupts through the stack-sharing code, the stack overhead seen by applications is limited to eight words (reduced to seven words for DOS 6, as the result of a slight revision to the code).

Configuring the Kernel

At the second configuration level are commands which act on the DOS kernel either to limit finite system resources or to prescribe behavioral aspects of the int 21h interface. As an example of the latter, setting **break=on** makes almost every DOS function check the keyboard for Ctrl-Break and the current standard input handle (which is not necessarily the keyboard) for Ctrl-C, Ctrl-P, and Ctrl-S.

Specifying a **country** and associated code page affects not only those few DOS functions dedicated to national language support but all file-related functions as well, because of implications for uppercase conversion when comparing filenames.

Finite Kernel Resources

Most commands at this level are concerned with structures used internally by the kernel to manage the file system. Poor configuration of these by users may result in performance degradation or inability to carry out file operations.

Performance is the **buffers** statement's claim to existence. Just to find a file on disk, let alone to access its contents, the kernel must set aside an area into which it may load disk sectors, especially from directories and the File Allocation Tables (FAT). Some temporary sector storage is also needed to support read/write requests for fewer bytes than contained in a whole sector. Efficiency dictates that the storage area be capable of holding most of the sectors needed to complete a typical file operation, otherwise much time will be wasted rereading sectors that had been in memory only moments earlier. This effect may be noted by rebooting with **buffers=1** and counting the minutes taken to run CHKDSK on a medium-sized hard drive. With too many buffers however, the kernel might easily spend more time searching among the buffers than it would take to reload a sector from disk; somewhere between the two extremes lies a number which is optimal for the particular operating conditions and which the user has some chance of guessing correctly.

When a file is opened, much of the relevant directory and FAT information that passes through the buffers is retained in a System File Table (SFT), along with various flags, counters, and pointers. Two independent sets of these tables are maintained to support the two different methods by which application programs may work with files. In handle-based file management, programs are returned nothing more than a handle that corresponds indirectly to an SFT. The number of these SFTs is limited by the **files** command. Although this indirection through a handle lends both security and invisible support for non-FAT file systems, it also means that whenever the supply of SFTs is exhausted, the system must fail all further requests to open or create a file using the modern handle-based functions.

The **fcbs** command specifies the number of SFTs used to support an older method of file access (inherited from CP/M and generally disregarded as obsolete but still used by several key DOS utilities, including COMMAND.COM). In this scheme, applications provide a structure called a File Control Block (FCB) to which DOS entrusts data describing the open file. DOS provides a matching SFT-like entry called a System FCB or FCB-SFT. In some circumstances, but not in the presence of file sharing or networking, these FCB-SFTs appear to be inexhaustible. If another is needed but all are in use, DOS may simply discard the one used least recently, safe in the knowledge that it can be reconstructed from data in an application's FCB.

Another limited resource is the number of drives. Some file systems such as UNIX accommodate drives in a single directory tree, paying no attention to the possible use of different physical storage devices. DOS is not quite able to do this: In the simplest configuration, each block storage device corresponds to a different drive supporting a separate directory tree. Beyond this, additional drives may be configured (with **join**, for example) that behave as, or at least like, directories—perhaps even directories belonging to an alien file system on another computer. To allow this interpretation of drives as logical entities not strictly related to physical storage devices, DOS maintains a Current Directory Structure (CDS) for each drive. Like the SFTs, these are a fixed resource: Their number is specified by using the **lastdrive** command to nominate the highest drive letter that will be required for the session.

Are FCBs Really Obsolete?

The int 21h FCB functions for deleting, finding, and renaming files (11h, 12h, 13h, 17h, and 23h) pass parameters in FCB format but do not require FCB-SFTs because they do not leave behind an open FCB. In this sense, they are merely FCB-style functions, and they continue to be used by many standard DOS utilities and internal DOS commands. Volume labels, however, seem surprisingly difficult to handle without recourse to the other functions in the superseded FCB interface: Both the FORMAT and LABEL utilities need an FCB-SFT to create, rename, or delete a volume label on pre-DOS 4 media.

That the interface actually is falling into obsolescence may be surmised from the minimal (and decreasing) support given to FCB functions by the DOS extenders built into Windows. In both versions 3.0 and 3.1, only functions 11h, 12h, 13h, and 17h receive support from the 32-bit DOS extender in the DOSMGR virtual device driver (VxD) for Windows enhanced mode. In Windows 3.0 standard mode, the 16-bit DOS extender program DOSX.EXE supports all FCB functions, but version 3.1 restricts this to just the four functions noted above (none of which, remember, actually open an FCB-SFT). A quick summary of the situation in Windows 3.1 standard mode DOSX is that functions 0Fh, 16h, and 23h are failed explicitly. The other "true" FCB functions are not supported at all since they will ordinarily not be called once the attempt to open or create the relevant file has failed; this measure of good behavior was not always observed by DOS applications that used FCBs for file I/O.

An easy, if undocumented, way to determine whether any software requiring FCB-SFTs has been run during the session is to inspect the word at offset 0010h in the DOS kernel's data segment. In general, this variable stores the number of times that application-based FCBs have been updated from system-based FCB-SFTs. Maintaining this count helps DOS with its Least Recently Used (LRU) management scheme for FCBs—which has, incidentally, been overhauled for DOS 5.

System-wide A20 Implications

Starting with DOS 5, the two system files and many standard DOS components are able to execute some of their code in the HMA—as noted in the introduction to this chapter. This ability carries implications for the design of these components, for memory layout, and for program behavior. By no means are the effects localized to the DOS kernel, which is why this configurability option itself is not discussed until a little later. Instead, the attention here is on a special accommodation made by the DOS kernel for older software that was either badly written or could not reasonably have anticipated the nature of this advance.

Without meaning to get too far ahead, observe that configuring DOS to run in the HMA implies that software does not rely on 8086 addressing compatibility. The software can tolerate the DOS kernel enabling the A20 as a normal consequence of calling the int 21h interface (and some others). Problems arise when programs that are unaware of this try to address memory in the first 64K by using segments > F000h (usually having subtracted past zero—a surprisingly common practice). DOS provides several remedies for these problems. Some are built into the DOS kernel and activated automatically. Others, such as the LOADFIX utility, are provided should the user discover a need for them. Provision for Windows/386 and Windows 3.0 Enhanced Mode takes the form of a file named WINA20.386, which Windows must load as a virtual device driver, but which is really a collection of routines that fix known problems as Windows is starting up (see below). By default, it falls to the DOS kernel to notify Windows about the WINA20 file's existence by hooking the Windows int 2Fh function 1605h initialization broadcast. To simplify matters, DOS requires that the file be in the root directory of the boot drive, but a **switches /w** statement in CONFIG.SYS relieves the DOS kernel of responsibility for installing WINA20, which must then instead be loaded through an explicit **device** entry (perhaps with a path to an alternative location) in the **[386Enh]** section of the SYSTEM.INI Windows initialization file.

WINA20.386

Although the means by which Windows creates a protected mode operating environment on top of real-mode DOS are beyond the scope of this book, the role of WINA20.386 in DOS 5 needs more explanation than is usually provided.

WINA20 does not install if the Windows version is 3.10 or greater; the file may safely be ignored or deleted if compatibility with earlier Windows versions is not needed. For these earlier Windows versions, the virtual device acts nontrivially if DOS 5 (or higher) is in the HMA or if the PSP of the WIN386.EXE program somehow lies in the first 64K.

Only when DOS is using the HMA does the virtual device hook into a chain of XMS handlers in order to hide a fixed number of A20 operations. This is roughly analogous to the way that the automatic enabling of the A20 implied by DOS's use of the HMA is delayed to support several classes of DOS programs that would otherwise not run properly. More detail about the DOS aspect of this problem is given in Chapter 5 under the heading "The Execution State."

The bulk of the WINA20 file deals with a problem of paging into the first 64K on certain 80386 machines. This facet of the device's behavior is controlled by profile strings

```
NoLow64KPaging=<Boolean>
```

and

```
NoLow64KPagingPort=<hexadecimal port address>
```

in the **[386Enh]** section of SYSTEM.INI. By default, paging into the first 64K is permitted on all 80486 machines and on 80386 machines that have no trouble reading a byte from the designated port (which defaults to 20h) using a REP INSB instruction. This is presumably to test that the CPU's paging unit handles the distinction between I/O and memory address spaces correctly. In Windows 3.1, this problem is handled by the virtual memory manager (VMM).

Whether or not a fix is required, WINA20 hooks the _Allocate_Global_V86_Data_Area service defined initially by the VMM. If paging into the low 64K is attempted but not allowed, then the installed hook obtains a dummy allocation of global space in the first 64K before letting the initial request proceed. This is roughly analogous to the way that LOADFIX forces its child program to load above the first 64K by first expanding its own memory allocation to cover the 64K boundary.

Additionally, if **NoLow64KPaging=true**, explicitly or by default, then WINA20 may install run-time patches to various VMM service routines that appear to be specific to Windows/386. This too has a rough analogy to a DOS fix, whereby certain programs in known formats also receive run-time patches to correct problematic code before being allowed to execute. Again, for more details on this DOS fix, refer to "The Execution State."

Installable Device Drivers

The third level of configurability extends the user an opportunity to provide device drivers for hardware that is either not known to the BIOS or is handled already by the default device drivers but not well enough to meet the user's expectations.

Hardware Not Known by Default

Almost by definition, the former class is not well represented on the DOS distribution disks. Peripheral devices common enough to warrant support from DOS "out of the box" would receive it in the form of a device driver embedded in IO.SYS; otherwise, the device driver should accompany the hardware.

The one standard example in this class of device drivers that provide DOS with the means to use hardware with no BIOS support is RAMDRIVE.SYS, although in its case the hardware used for storage is just more of the computer's RAM—preferably main memory outside of DOS's normal range (a time-honored use of extended memory), but the driver can also use expanded or conventional memory.

Since DRIVER.SYS enables DOS's use of external floppy disk drives, it also belongs in this class of device drivers for hardware not under DOS management by default. In another sense however, DRIVER.SYS merely extends the existing BIOS support: The external drives are operated through int 13h functions and indeed, through the default block device driver in IO.SYS, just as if their existence had been reported by the BIOS during IO.SYS initialization.

Enhancing the Default Support

Device drivers that enhance access to hardware already controlled by DOS figure much more prominently in the collection of installable device drivers packaged with DOS on the distribution disks. The most familiar example is probably ANSI.SYS, which replaces the default CON device and enables easy variation of the usual teletype style of screen output by interpreting certain escape sequences in the character stream as directions to change the color or location for the next character, etc. Recent versions offer an I/O Control (specifically, a Generic IOCTL) interface through which programs may interrogate the driver about the current video configuration and perform operations such as changing the number of text rows—admittedly for standard video modes only—without having to confront the wide variety of video hardware and the attendant complications.

DISPLAY.SYS continues this notion of adding features to the default CON device. However, it acts exclusively as a control device: Unlike ANSI.SYS, it has no direct effect on the character stream; instead, applications may use a Generic IOCTL interface to have the device driver reprogram the video controller's character generator using fonts from a .CPI file sent as device-specific control data. PRINTER.SYS works similarly to support code pages for the LPT family of default device drivers. Often, but not necessarily, this support for multiple character sets (code pages) operates in conjunction with DOS's national language support. For instance, the numbers 155 and 157 correspond to the characters ¢ and ¥ in the code page usually associated with the United States but are displayed as ø and Ø if fonts have been loaded for the Scandinavian code page.

Code Page Support

Code pages have been a feature of DOS since version 3.30, but even in Europe, where the occasion for their use arises frequently, the DISPLAY and PRINTER device drivers remain a mystery to both users and programmers. When texts on DOS programming list int 21h functions, it is quite common just to acknowledge the Generic IOCTL interface for code-page switching but omit details. Both the interface and the .CPI file format are documented in the official reference for DOS 5 and 6. That said, Microsoft could save users and programmers a lot of trouble, if not actually promote code page support, by ensuring that its interface behave consistently and that its technical documentation match reality.

For an example of inconsistency visible at user level, consider that **mode con cp select=*nnn*** generates an error message if the code page desired for DISPLAY is incompatible with the current KEYB installation but changes the display's character set nonetheless. The inconsistency is that in this same situation, **mode con cp refresh** generates the same error message but *does not* affect the displayed character set.

Following the official literature might lead a programmer to think that calling Start Code Page Prepare (int 21h function 440Ch minor code 4Ch) with all code page numbers set to FFFFh will succeed in refreshing the current code page (for instance, to recover from software that loaded a custom character set but terminated unexpectedly). Pursuing this line would waste the programmer's time, energy, and diminishing good will, since this recommended method produces nothing but an error, at least in the current implementation of DISPLAY.SYS. (The current code page may be refreshed, however, by passing the same function a count of zero code pages.)

Device Drivers of Convenience

The ANSI.SYS and DISPLAY.SYS device drivers also trap int 10h either to keep reasonably well-informed of the true hardware state or to impose some sort of consistency between programs that use DOS for hardware control and programs that side-step DOS by calling BIOS functions directly. The EGA.SYS device driver hooks int 10h in an attempt to maintain a picture of the video controller's registers, most of which are read-only on the EGA. Knowledge of the current values in these registers is vital in task-switching environments, where the video state constructed by one program must be saved and restored by another. Programs that know of the EGA Register Interface may read from and write to video registers through extra int 10h functions. Given that this facility is obviously hardware-related, it is quite natural that its implementation be presented as a device driver, but in fact, the EGA$ device does not channel a character stream between the DOS kernel and some piece of hardware: It exists only to support the EGA Register Interface and its effects could be duplicated by a non-driver resident program.[1]

Indeed, many programs loaded through **device** or **devicehigh** directives (the latter being a DOS 5 variation to place the device driver in upper memory if possible) are not device drivers in the formal sense that they lend DOS the means to operate hardware

---

[1] A reduced implementation of the EGA Register Interface is provided by the Microsoft Mouse driver—not supplied as standard with DOS. If EGA.SYS is installed beforehand, then the mouse driver knows not to retain the code dedicated to this interface (slightly more than 1K).

(whether physical or emulated). Other examples of these "device drivers of convenience" on the DOS distribution disks are EMM386, HIMEM, SETVER, and SMARTDRV. These are device drivers in order to load the software as early as possible. System startup is arranged so that lower levels of configuration are settled before building further. Therefore, the initialization command for an installable device driver is the first opportunity to execute code selected by the user.

HIMEM.SYS, which installs XMS routines for HMA management, A20 control, and extended memory access from real mode, exists as a device driver for no other purpose than early execution. It provides only a device driver initialization function and could easily be recoded as a normal program. Readers with access to the Windows 3.1 distribution disks may find a program called XMSMMGR.EXE, which is nothing but HIMEM.SYS in executable program form. (This program is needed by Windows SETUP, now that **win /r** is gone, in case the user doesn't have an XMS server installed.) The equivalence of resident programs and device drivers that care only for their initialization is probably more familiar to readers who have the Microsoft Mouse driver—distributed both as a device driver MOUSE.SYS and as a resident program MOUSE.COM.

DOS 5 documents the facility introduced in DOS 4 for misleading programs about the DOS version number. A cynical expression of the idea behind this is that recent leaps in major version number, in return for arguably smaller increments in functionality than programmers had come to expect when the industry was enjoying rapid expansion, have taken much software by surprise. Some programs fail under DOS 5 simply because their designers did not anticipate such large steps in DOS version to occur so quickly. Insofar as the scheme works only for well-behaved software (or for programs whose rule-breaking has been accommodated specially in the design of later DOS versions), SETVER.EXE can be used to make DOS 4.01 seem like 3.40 or DOS 5 like 4.10, etc.—to the version-checking routines of named applications as well as to disaffected users. As with HIMEM.SYS, this facility may be needed very early in the system's startup sequence. SETVER is therefore coded as a device driver but implements no element of the device driver interface other than the initialization function: All it need do is install its version table, ready for the DOS kernel to consult whenever a new program is loaded. Unlike HIMEM.SYS, however, the "device driver" may also be run as a program in the usual fashion—in this case, to interrogate or alter the version table in the SETVER.EXE file.

SMARTDRV.SYS hooks int 13h to intercept attempts to read from or write to hard disks. It builds a cache of recently used tracks so that time-consuming disk access can be avoided in favor of retrieving a track from extended or expanded memory. Counteracting this advantage is the overhead of maintaining cache integrity when data is written to disk. Again, the facility is cast in the form of a device driver to allow early use. Moreover, the opportunity is taken to use the device driver IOCTL interface as the means for software to control the cache's operation; this has a certain precision in comparison with interrupt-based methods, since the device must be opened by name (substantially reducing the possibility of accidental conflict). On the other hand, SMARTDrive 4.0 and higher can be run as a normal executable and provides an int 2Fh interface instead of the older IOCTL interface.

By far the most complicated program loaded as a device driver and supplied as standard with DOS is EMM386.EXE. It constructs a protected-mode environment in which the DOS session runs as a virtual-8086 task. Careful use of the 80386 paging mechanism produces a situation in which pieces of memory from all over the 80386 address space are stitched together for the task—to be addressed as if the processor were still in 8086 real mode. Other operating systems hosted on 80386 machines provide one or more wholly independent DOS sessions, each with its own megabyte (plus HMA) of addressable memory containing a separately booted copy of DOS. Alternatively, a system like Windows creates virtual-8086 tasks which share important regions of memory, such as the DOS kernel and ROM BIOS, avoiding duplication at the expense of protection. The supervisory system must "instance" sensitive memory in the hope of preventing one virtual machine from corrupting the others, but the advantage is that a given number of "DOS boxes" requires less memory in this scheme. With EMM386, there is only one virtual-8086 task to consider and therefore fewer complications. Although the protected mode EMM386 is certainly in charge of the computer, its intention is not to run the DOS session as a client but to provide DOS with a useful service—specifically to supply memory at addresses where it is not physically present. This additional memory is ideal for loading resident material away from the 640K of conventional memory, so typically EMM386 or some similar memory manager will be one of the first pieces of user-specified code executed in the session.

As with SMARTDRV.SYS, EMM386 uses DOS's provision for interfacing with device drivers, even though it is not associated with a character stream or block storage. Thus, EMM386 responds to a variety of IOCTL requests (and one in particular that all memory managers must offer if they are to co-exist with Windows enhanced mode). In very much the same way that a TSR may be re-run in order to interact with a resident copy of the program, EMM386.EXE may be executed as a normal program after its installation in order to configure the resident driver or report its status.

In practice then, the distinction between device drivers for low-level hardware control and programs for higher-level applications blurs a little. Table 1-1 summarizes how some of the device drivers of convenience supplied with DOS cover a range of possibilities.

Table 1-1: DOS-Supplied Device Drivers

USES DEVICE DRIVER INTERFACE FOR PURPOSES OTHER THAN INITIALIZATION	CAN ALSO RUN AS A PROGRAM TO COMMUNICATE WITH RESIDENT DRIVER	EXAMPLE AMONG DRIVERS DISTRIBUTED AS STANDARD WITH DOS
no	no	HIMEM.SYS
no	yes	SETVER.EXE
yes	no	SMARTDRV.SYS
yes	yes	EMM386.EXE

One of this book's goals is to show how analysis of the system's startup and configuration leads naturally to a programming framework in which the device-driver and application program formats are equivalent—so much so that device drivers can be written in C with the usual `main` function serving as the entry point to device driver initialization and with safe access to standard library code. Indeed, a large class of programs (resident or not) may be converted to device drivers of convenience simply by relinking their object code with the device driver library developed in Chapter 7.

Running Programs

System configuration may reasonably be thought complete when the system is ready to execute programs, but just as it is possible to see some device drivers as specially-coded TSRs, it also makes sense to regard some resident programs as being part of the system rather than part of the user's operating environment. DOS provides the means to load TSRs as part of the system's general configuration through the **install** command. This point is moot in DOS 5; it appears to have been introduced in DOS 4 only as the means to force installation of SHARE in the presence of hard disks with more than 64K sectors, a requirement that no longer applies (see below). Programs that run at this stage receive an empty environment (that is, one that may be consulted for the program pathname but that defines no environment variables); they therefore retain less memory or leave a smaller hole in the memory map than they would if run later. Another difference between program execution at this stage and after the system is fully configured is that only minimal handlers are in place for the DOS exceptions; thus, Ctrl-C will not terminate the program unless it specifically sets up its own int 23h handler, and, likewise, all critical errors are failed automatically.

DOS 4 and SHARE

DOS 4 required SHARE in the presence of a large hard disk to overcome a compatibility problem with the FCB method of handling open files. Since FCBs are maintained in program memory rather than by DOS itself, expansion of the FCB to record logical sector numbers > 64K, say by a simple widening of fields from 16-bit to 32-bit, would certainly break existing applications. On the other hand, leaving the FCB structure unchanged meant that information about an open file whose directory entries lay more than 64K sectors into a disk partition could not be recorded properly in an application's FCB. This is no problem as long as DOS has the information in an SFT, but clearly the normal practice by which DOS reuses the SFTs dedicated to FCB support cannot be sustained. The solution in DOS 4 was to load the file sharing extension, which disables FCB-SFT reusability.

It is of course a bit excessive to force installation of a whole program for this purpose, even if the program happens to be one like SHARE, whose use is strongly recommended. DOS 5 avoids the problem altogether by reorganizing the reserved portion of the FCB structure in a way that does not also require an expansion. Because cluster numbers are still limited to 16 bits, a 32-bit field for logical sector numbers is actually extravagant; even with 128 sectors per cluster, room need be found only for seven more bits. For a slight performance loss in an obsolete interface, one of the reserved FCB fields previously used to optimize the regeneration of FCB-SFTs is instead used for the high six bits of the directory entry's logical sector number. Yes, that's six bits, not seven, meaning that the FCB interface breaks down at more than 64 sectors per cluster.

System configuration certainly comes to an end with the launch of a command processor, which creates the user's preferred operating environment. By default, the **shell** is a copy of COMMAND.COM in the root directory of the boot drive. It generates a text-based interface with a prompt to which users respond by entering the names of programs to run or commands such as **date** and **time** that COMMAND.COM handles internally. The main requirement for a shell is that it must never terminate: The machinery of system configuration disappears when the shell is started (while SYSINIT in DOS 6 goes to some trouble to guard against a missing shell, this is not quite the same as guarding against shell termination). In addition, it must be understood that the shell is executed without an environment; not only are there no environment variables, but there is no program pathname and indeed, no environment segment. Thus, even if one were tempted to use the MS-DOS Shell or Windows for this purpose, thinking that these programs' exit options could be disabled or avoided, it would be unsound. Both DOSSHELL.COM and WIN.COM assume the prior existence of an environment, without which they are at best unable to bootstrap the remainder of the MS-DOS Shell or Windows (furthermore, starting either program without an environment may crash the system).

Four DOS Regimes

The **dos** statement in CONFIG.SYS—new to DOS 5—places the system in one of four operating regimes, with results that pervade the system, affecting almost everything from the structure and content of IO.SYS to the global memory arena in which programs are executed. The four possibilities are constructed by combining two essentially independent features. On the one side of this product, DOS may be either **high**, meaning that the High Memory Area (HMA) will be used for system components, or **low**. On the other side, DOS's **umb** management may be enabled or not (denoted by **noumb**). The default regime—the one compatible with earlier DOS versions—is the combination **dos=low,noumb**.

Managing Upper Memory

The **dos=umb** option aims to provide programs with memory in the area above conventional memory but still inside the one-megabyte 8086 address space. How this extra memory happens to be fitted, remapped, or otherwise created out of ether is of no concern to DOS or to this book (though certainly, a close study of EMM386 makes a fascinating introduction to protected-mode programming and particularly to some of the problems that confront the larger-scale DOS virtualizations in Windows). DOS's role is only to take over responsibility for this upper memory so that it can be allocated to programs as part of the usual int 21h interface for memory management.

Configuring the DOS kernel to manage upper memory eases the loading of programs into upper memory specifically. An expanded collection of memory allocation strategies offers enough control for most practical purposes, using either the **loadhigh** command built into COMMAND.COM or some third-party replacement. Without **dos=umb**, upper memory is available only to programs that communicate directly with the UMB provider, and a substitute **loadhigh** requires at least some work with structures below the int 21h interface, plus careful planning if the program loaded into upper memory is to be able to allocate more memory during its execution.

As well as extending the range of the kernel's memory management services, setting **dos=umb** also enables the distinction between **device** and **devicehigh** statements in CONFIG.SYS. Device drivers are overlays and do not receive memory allocations of their own. The **devicehigh** mechanism is not easily reproducible like **loadhigh**; there is more to it than simply understanding the extra memory allocation strategies.

Using the High Memory Area

If **dos=high**, the High Memory Area (HMA) is set aside for the system's use. As noted in the introduction to this chapter, fairly large portions of IO.SYS, MSDOS.SYS, COMMAND.COM, and HIMEM.SYS are executed from the HMA. Indeed, almost the whole of the DOS kernel is loaded into the HMA, along with its **buffers** system. The remaining space in the HMA (which is only 64K in size) can easily be taken up with fonts prepared for DISPLAY.SYS. A code page with three subfonts for the VGA requires roughly 9K, quite an extravagance in conventional memory for infrequently used characters sets. That this relocation clears space in conventional memory is of course the most obvious consequence at user level. Indeed, so much extra space is made available that some programs fail when **dos=high** because they were not designed for execution so low in memory. Deliberately or not, they assume that the system will always occupy the first 64K. Now much of that first 64K may become available to device drivers and programs.

In addition to the global impact on the arrangement of programs in memory, setting **dos=high** has an important consequence for the A20, which must be enabled while the system executes in the HMA. To avoid a performance loss (for some hardware, A20 operations are very slow), most system interfaces enable the A20—even after the system call returns. The A20 is not enabled permanently, most notably because DOS disables it when a new program starts, but as a general rule, use of **dos=high** implies that no software run during the session should rely on memory addresses to wrap-around at 1MB.

ROM-Executable DOS

Because the HMA is effectively not present unless the A20 is enabled, the system cannot be moved to the HMA in its entirety. All system interfaces must have stubs in low memory that make sure the A20 is enabled before control is passed to the HMA portion (which handles the bulk of the processing). A similar argument means that all items of system data whose addresses are publicly available must also be in low memory; existing software will not know of any connection between addressability of system data and the state of the A20. Indeed, the system files are so rich in data liable to be accessed directly, that, effectively, the low memory portions become data segments and the HMA portions become code segments—which is often how they are referred to in subsequent chapters. This separation of read-only material (typically code, but also constant data) from read/write data may arise as a natural side-effect of DOS's use of the HMA but it is also the main requirement of running DOS in ROM.

Of course, the ROM-executable versions of the system files differ from the disk-based versions, most notably because the DOS kernel offers extra int 21h functions for finding program modules in ROM (see Chapter 2) and has variants of some others—for instance, to support the execution of these programs from ROM.

In particular, a good understanding of the disk-based DOS should be easily adaptable to the ROM form. There are many possible layouts of ROM-executable DOS. In order to boot this form of DOS, there must of course be some image in the first megabyte. When this ROM expansion module is initialized by the ROM BIOS, it hooks int 19h so that the BIOS's subsequent attempt to load an operating system from disk actually boots the ROM-executable DOS instead. For the rest of this DOS, there is much flexibility. The main property of the system files' and command processor's data segments is that a ROM-specific loader must copy these segments to RAM in low memory. It does not matter for this purpose whether the segments are taken from ROM inside the first megabyte or outside, though the former has the disadvantage of wasting some address space in DOS sessions that do not have a memory manager to map (virtual) RAM over the redundant data segments in (physical) ROM.

Although significant changes to the DOS programming interface are studied, this book concentrates mostly on the disk-based DOS implementation, especially when following the startup sequence in Chapter 3. Of course, the author is not alone trying to avoid having too many variations to follow coherently. A similar threat would have confronted DOS's designers, which is why the identification of concerns (public versus private for HMA execution and read/write versus read-only for execution in ROM) is so important: It leaves much less to think about in the way of special cases.[2]

---

[2] There exist only two exceptions to this goal of common treatment for objects that must be in both the disk-based and ROM-executable versions. The code segment in the disk-based IO.SYS contains one byte and one word that are written to, presumably as oversights. The byte is the drive letter in the message about inserting a disk when the logical drive map is changed; without special attention, the ROM version could only ever request a diskette for drive A. The word is used to index sectors in the root directory when searching for a volume label on a diskette that does not have a media ID in its boot sector; the price is that the ROM version must work around this in order to recognize old-style volume labels reliably.

CHAPTER 2

The System Footprint

For a practical understanding of **dos=high** and the benefits of moving system material to the HMA, it may help to inspect the system files and standard command processor to see just what savings are made. Microsoft has internally reorganized each file to ease the separation of material that can be relocated to the HMA from material that must stay in low memory. The run-time machinery for effecting this separation is examined in Chapter 3. The **devicehigh** statement enabled by **dos=umb** cannot be grasped properly without following the system's startup sequence. For this purpose and several more of the finer points of memory management under DOS 5 and 6, it is important not just to assess the scale of the reorganization but also to acquire some intimacy with the contents of the system files.

While DBLSPACE.BIN (which provides disk compression in DOS 6) is sometimes described as a new system file, DBLSPACE.BIN is not discussed until Chapter 3, which examines the DOS startup sequence. DBLSPACE.BIN is just a device driver that hooks into other device drivers, as is SMARTDRV.EXE (except that you run SMARTDRV as a program). It's an extension to the default block device driver in very much the same way that ANSI.SYS and DISPLAY.SYS replace the default CON device driver, except that with block device drivers, this replacement process is a bit more complicated.

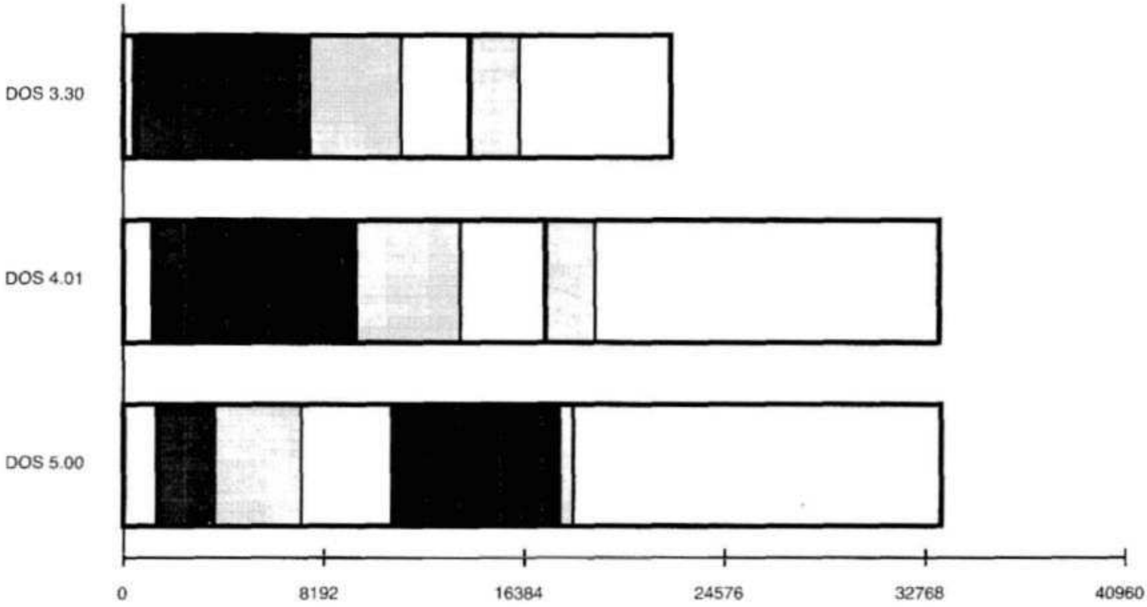
IO.SYS

It might seem that IO.SYS would contain just the default device drivers and hardware support routines, together with the necessary initialization code. The startup sequence would progress to the next system file—the DOS kernel—which would set up the main system interfaces before dealing with user-level configuration and the execution of a command processor.

Actually, the startup sequence is much more of a bootstrap than a chain. It has been observed already that provision exists for IO.SYS to be distributed in modified form to support DOS on machines that are not strictly PC-compatible. In contrast, MSDOS.SYS does not deal with the hardware-dependent aspects of the system and is essentially the same for all DOS machines. To support user-level configuration of nonstandard hardware while keeping to the pattern of two system files, control reverts from MSDOS.SYS to IO.SYS for the analysis of CONFIG.SYS.

Thus, IO.SYS is modular: One component provides the default device drivers and performs the automatic configuration before the system interfaces are enabled; a second module—usually called SYSINIT—initializes the DOS kernel, reads CONFIG.SYS, and launches the shell. In DOS 6, IO.SYS preloads the disk compression driver, DBLSPACE.BIN (see the end of Chapter 3). Successive DOS versions have added to this internal structure, as shown for the most recent cases in Figure 2-1 and the explanatory notes that follow it.

Figure 2-1: Structure of IO.SYS on Disk—Comparison of Different DOS Versions



Material retained in memory is shown shaded: lighter shading denotes that retention is only conditional. Thick borders indicate the file's segmentation. From left to right, in each version, the file is shown in the portions:

1. A small loader—introduced with DOS 3.30—consisting of contiguous sectors read into memory by code in the boot sector and knowing enough about the file system to be able to read the remainder of the possibly discontiguous IO.SYS file (having moved out of the way first!);

2a. The main IO.SYS module, divided in thirds for Figure 2-1 according to purpose and ultimate effect on memory layout:

> The main resident component, containing the default device drivers and various pieces of code that provide low-level support to the system as a whole;

> Some optional material—the amount retained depends on the number of floppy disk drives, hard disk partitions, the model of machine and BIOS, etc.;
>
> Discardable initialization code for IO.SYS—also responsible for loading the MSDOS.SYS file into memory;

2b. In DOS 5 and higher, a second segment of the main module may be relocated to the High Memory Area if **dos=high** is selected in CONFIG.SYS;

3. The SYSINIT module, also considered in pieces:

> Optional stack-switching support for hardware interrupts and the NMI—significantly reduced in size for DOS 5;
>
> The mostly discardable remainder of the SYSINIT module—configures the system according to directives in the CONFIG.SYS file and loads the command processor.

The IO.SYS Loader

The first portion of the IO.SYS file would be largely uninteresting if not for an explicit claim in the DOS manual (under the entry for the SYS utility) that previous versions' requirements of sector-wise contiguity for IO.SYS no longer apply. Were this claim entirely accurate, it would suffice to assume that these three sectors worth of code had already done their work and loaded the rest of the file into memory. Instead, the reader is invited to take the first of this chapter's many lengthy asides prompted partly by the author's undisciplined, digressive thinking style but roundly encouraged by the frequent disparity between Microsoft's documentation and reality.

The idea behind the IO.SYS loader is that the boot sector—a mere 512 bytes—is too small to carry the task of loading the system files. Instead, the boot sector need only locate the root directory, verify that the first two entries correspond to files named IO.SYS and MSDOS.SYS, respectively, and then read a few sectors worth of code from IO.SYS that will in turn load the rest of that file; fetching MSDOS.SYS from disk need not be contemplated until much later. A boot sector left to load the system files on its own—as happens in DOS versions before 3.30—can apparently not also follow their distribution through the File Allocation Table. As a consequence, early DOS versions imposed strict conditions on the two system files: Not only had they to lie first and second as directory entries but also as files on disk—sector by sector, without a break.

This contiguity constraint is relieved by having the IO.SYS loader trace the cluster chain. More space becomes available in which to program this operation, but the sectors comprising the loader itself must still be contiguous if they are to be retrieved from disk correctly by the boot code. In DOS 5, this is the only remaining contiguity requirement: IO.SYS and MSDOS.SYS may lie anywhere on disk, in any number of pieces, as long as the first *three* sectors of IO.SYS (the size of the loader portion) are together.

At least to the author's mind, this caveat does not figure in the manual's claim that "MS-DOS no longer requires the two system files to be contiguous." Perhaps this is open to interpretation, but it seems to leave only the condition that IO.SYS and MSDOS.SYS be the first two entries in the root directory. The practical effect should be that attempting to boot with a disk whose system files had been transferred using the **copy** command would either be successful (if by chance or calculation, the first two root directory entries had been empty) or would be handled safely through a warning such as

```
Non-System disk or disk error
Replace and press any key when ready
```

However, a remaining contiguity assumption exposes DOS to the possibility of a bootstrap failure that is not caught as an error by the boot code, the most likely result being a system crash at startup. Any inference that this has been remedied in DOS 5 is incorrect. On a disk with one or two sectors per cluster, it is possible—using normal DOS commands—to transfer the system files in a way that leaves a discontiguous IO.SYS loader and creates a "crash disk." It seems likely that someone at Microsoft knows of this problem, whatever is stated in the manual, because the SYS utility always arranges three contiguous sectors for the beginning of IO.SYS on any startup disk it creates.

Problems with SYS

Incidentally, the SYS utility contains some strikingly shoddy programming. Readers with a disk editor may care for the following safe, easy demonstration. Take an empty floppy disk and edit the FATs. At cluster 2, place the carefully-contrived value 247. At cluster 247, place an end-of-file marker. Write the changes to disk, updating both FAT copies. The result is two clusters in a single chain. Use **chkdsk /f** to turn this into a file. The construction may seem unusual, but the result is a perfectly valid disk on which two clusters of specific interest are allocated to a file—a situation which can occur naturally. The SYS utility (at least in later DOS versions) is ordinarily untroubled by a non-empty disk with so much free space. Nevertheless, attempting to transfer the system files to this disk, by running something like

```
sys c:\ a:
```

produces the error

```
No room for system on destination disk
```

The experiment succeeds if repeated with 246, 248, or indeed almost any value other than 247, because SYS thinks cluster 2 on the target disk is bad; yes, 0FF7h is the marker for a bad cluster (or FFF7h if the disk has 16-bit FAT entries), but when SYS looks for free space beginning with cluster 2, it checks only the FAT entry's low byte for a possible bad cluster marker—apparently forgetting that for a sufficiently large disk, F7h (the hexadecimal expression for 247), 01F7h, 02F7h, etc. are all valid cluster numbers.

Starting with DOS 5, the SYS utility also copies a command processor to the newly-created system disk—meaning that for the first time, applying SYS to a blank disk actually produces something bootable. This is welcome, but be warned that the utility does not necessarily take the command processor from the same source as the system files. If no source path is given explicitly, then SYS does indeed take the system files IO.SYS and MSDOS.SYS from the root directory of the current drive—as described in the manual. If, however, the current drive is also the one from which the computer was booted and the COMSPEC environment variable designates COMMAND.COM as the command processor, then SYS attempts to take COMMAND.COM from the COMSPEC path first, in preference to the root directory of the current drive. (FORMAT /S, which has been reported for example to copy 4DOS.COM to disks using the name COMMAND.COM, is a different story.)

Low and High Segments

In DOS 5, the main IO.SYS module—the one that provides the default device drivers and other hardware support—is split into two separately addressable segments. The intention is that the second segment be moved to the HMA, where its memory consumption will be less of an imposition. Figure 2-1 shows that this second segment accounts for most of the IO.SYS footprint. The first segment consists of everything left behind after subtracting resident material suitable for the HMA and is essentially a reduction of the whole module from earlier DOS versions. There is also a small overhead to support exchanges between the segments: executing code in the HMA always requires support in low memory.

The lines of division for any program contemplating use of the HMA are formed by two constraining properties of this memory region:

- It is common practice in DOS to express addresses in a canonical form with offset less than 0010h. HMA addresses cannot be standardized this way and should not be passed to software that assumes such pointer-normalization reductions to be possible. Notable cases are DOS itself for file I/O and many BIOS implementations of the int 13h interface for disk access.
- The HMA is visible only while the A20 line is enabled. Addresses in the HMA should not be communicated to software that might be unaware of the A20 state. A practical implication noted earlier is that publicly accessible data must be in low memory. Another is that if code in the HMA is liable to be executed when the A20 may be

disabled, then the code must be entered from low memory so that the A20 can be enabled first.

Of the two constraints, the latter has the wider effect; using an address ≥ FFFF:0010 but misunderstanding the A20 state leads to two types of error:

- First, the intention may be to access the HMA, but the attempt goes astray because the A20 is disabled and instead wraps around to accidentally access memory somewhere in the first 64K. Preventing this error is mostly the responsibility of the HMA owner, which must ensure that other software may access material in the HMA only when the A20 is known to be enabled. In all circumstances where this is not certain, the HMA must be hidden behind an interface in low memory. HMA owners should be aware that this principle is violated implicitly if their HMA code calls through an interface that may disable the A20, since the address pushed onto the stack for a far call or software interrupt (thus making an HMA address available externally) will be misinterpreted when the call returns. To prevent this, HMA owners must divert control to low memory when calling software that cannot be trusted to maintain the A20 state. Whether or not the A20 should be disabled while issuing the external call from low memory depends on the HMA owner's attitude to the second type of HMA addressing error. Observe that nothing can be done to overcome a (faulty) hardware interrupt handler that disables the A20.
- In the second error type, an address seemingly ≥ FFFF:0010 is meant to access memory in the first 64K, but goes to the HMA instead, because the A20 is enabled. The HMA owner can do little to detect this error, let alone prevent it. When the HMA owner is DOS itself, choosing a **dos=high** regime implies (in general) that this type of error will not occur. Necessarily, this assurance must come from the user: DOS cannot know the mind of a programmer who prefers an address ≥ FFFF:0010 for low memory.

Applying these considerations to the particular case of IO.SYS (explained earlier to be rich in data that is publicly accessible, though not often through documented interfaces) produces an overall picture in which the segment destined for removal to the HMA consists almost entirely of code, and, conversely, the segment left in low memory is essentially a data segment.

The strategy and interrupt routines of the default device drivers are necessarily in the same segment as their corresponding device driver headers, since the header specifies only the offset of these routines. Since these headers are publicly accessible data (used, for example, by every program that walks the DOS device chain; the programs would not know to first check A20), they must be in low memory, so for the default device drivers, the stubs which hide the HMA code when **dos=high** exist naturally. Indeed, the common strategy routine lies entirely in low memory, since it contains too few instructions to justify the overhead involved in transferring control between segments. For the interrupt routines, the objective is to switch execution to the HMA at the least possible cost. If the high segment is in conventional memory rather than in the HMA, then segmentation is just an inconvenience requiring a far jump; otherwise, it is essential to ensure that the A20 is enabled before proceeding. Most probably, the device driver request will have come from the DOS kernel—also executing in the HMA—so that as long as the device driver request packet lies in the DOS kernel's data segment, the A20 is judged to be enabled already. If the paragraphs at addresses 0000:0080 and FFFF:0090 differ, the A20 is already enabled. Otherwise, IO.SYS asks the XMS server to enable the A20 line—a hardware-dependent operation for which the XMS provides a general interface (see Chapters 9 and 11).

Note the trouble taken by IO.SYS to test the A20 state rather than simply issue the relevant XMS call. A guiding principle in matters of A20 control is that the A20 hardware should be used sparingly because it is very slow on some machines. The decision is usually handled by the XMS server, whose implementations of Local Enable/Disable A20 functions maintain a counter and query the status of the line to avoid redundant hardware access. Like any XMS client, IO.SYS could simply issue a Local Enable A20 call to gain access to the HMA, then match this with a Local Disable A20 call on completion, knowing that the logistics of A20 control are safe in the hands of the XMS server. IO.SYS, however, does not disable the A20 and must not confuse the XMS server's internal accounting. For more information on the A20 and its management through the XMS, refer to Chapter 9; the important point here is that if IO.SYS has to enable the A20 for a device driver call, then, in keeping with DOS's normal attitude to the A20 when **dos=high**, the A20 is left enabled on completion of the device driver request.

As with the device driver strategy routine, most of the interrupt handlers in IO.SYS are sufficiently small to be left in the low segment. This is trivially true for interrupts 01h, 03h, and 04h, which are all pointed to an IRET instruction. Not far behind in simplicity are the handlers for int 1Bh (which simply records the Ctrl-Break combination of keystrokes as a Ctrl-C character for the default CON device driver) and int 29h (which provides the default CON device driver with its teletype-style output via int 10h to the video system).

All other interrupt handlers in IO.SYS are more like filters, in the sense that they supplement services offered by other programs (typically the ROM BIOS) rather than provide new ones. Two of these—int 19h and (starting with DOS 5) int 15h—are also small enough to be dealt with entirely in low memory, but carry implications for the performance record of the programming industry, which we will now discuss at some length.

Int 19h—Preparing for a Warm Boot

When the BIOS reloads the operating system in response to an int 19h (Reboot) instruction, it does not reset the Interrupt Vector Table or restore any other system resources to their BIOS defaults, except perhaps to ensure that the entry for int 1Eh points to a valid Diskette Parameter Table. This exposes the new warm-booted session to the possibility that an interrupt vector points to an address which was valid before the reload but has since been overwritten with new material. Should this be the case, the consequences of issuing this interrupt in the new DOS session are unpredictable, but the most likely result, if preventative measures are not taken, is that the computer crashes as the system tries to rebuild. In general, *all* software that hooks interrupt vectors must also intercept int 19h so that the vectors in question can be restored before the request to reboot reaches the BIOS. Fortunately, there exist two large classes of exemptions:

- Responsibilty for restoring a given vector lies only with the first program that hooks the interrupt from its BIOS default. Since IO.SYS is the first program to run during the session, it cannot escape through this provision and must restore all the vectors it hooks. Those interrupts are removed from the problem domain for all other software loaded during the session.
- It may be that a fresh handler is guaranteed to be installed from scratch before the interrupt's first use by the incoming warm-booted DOS session. This is presumably the case with interrupts 20h to 3Fh inclusive, which are re-installed each time DOS starts. In practice, this consideration also applies to int 00h, since it too is handled afresh by the DOS kernel; note however, that this particular case's soundness relies on the boot code and system file initialization to be free of divide-by-zero errors (at least until the new handler is installed). Similar considerations apply to interrupts 01h, 03h, and 04h.

Since the only two programs certain to be run during a DOS session are the two system files, blending these considerations with knowledge of the interrupt vectors handled by IO.SYS and MSDOS.SYS or explicitly restored by IO.SYS when int 19h is intercepted leads to the list in Table 2-1—the interrupts for which software may assume that precautions for int 19h are either in place already or unnecessary.

Table 2-1: Interrupts Exempted from Int 19h Precautions

INTERRUPTS	REASON	CONDITIONS
00h	re-installed in new session	
01h	re-installed in new session	
02h	restored before reboot	if **stacks** are configured
03h, 04h	re-installed in new session	
08h, 09h	restored before reboot	if **stacks** are configured

Table 2-1: Continued

INTERRUPTS	REASON	CONDITIONS
0Ah to 0Eh	restored before reboot	if **stacks** are configured*
0Fh	re-installed in new session	only if vector lies below segment A000h or in the ROM BIOS segment F000h+
10h	restored before reboot	starting with DOS 5
13h	restored before reboot	
15h	restored before reboot	starting with DOS 5
19h, 1Bh	restored before reboot	
20h to 3Fh	re-installed in new session	
6Ch	re-installed in new session	only for PC Convertible with System Resume feature—to update DOS clock after suspension
70h	restored before reboot	if **stacks** are configured
72h, 73h, 74h, 76h, 77h	restored before reboot	if **stacks** are configured*

* *These interrupts are not hooked for stack-sharing (and therefore are not restored before the reboot) if they seem to act trivially (e.g., are handled simply by an IRET instruction) when the* ***stacks*** *support is set up.*

+ *This vector must therefore be restored before the reboot if it has been hooked in upper memory.*

One way to view this list is that it leaves the restoration of most interrupt vectors in the hands of software. In a programming environment built on cooperation, such incompleteness is no problem as long as programs understand their duty with respect to int 19h and can be trusted to perform. Another way to regard the list is that it covers the interrupts most likely to be hooked, so the responsibilities imposed on TSRs are far from sweeping.

Despite this, int 19h may as well be written off as an interface for reloading the operating system. In many of the different configurations possible for a DOS session, executing int 19h will almost certainly crash the computer, even if the only resident programs are standard DOS utilities distributed by Microsoft.

Consider the interrupts hooked by specific applications. Notable omissions from Table 2-1 are interrupts 14h and 17h, hooked by MODE to redirect printing from parallel to serial ports. When the resident part of MODE is installed for this purpose, it is responsible for intercepting int 19h so that it can restore the other interrupts when the system is rebooted. Its failure to take this precaution renders int 19h unsafe if MODE is installed. The same is true of the GRAPHICS utility, which traps int 05h (invoked when the keyboard handler encounters Shift-PrintScreen). Another vector often taken over by TSRs but not in the list is int 1Ch (Timer).

Some memory managers defeat int 19h because of their failure to restore interrupts 4Bh or 67h (which are hooked for the Virtual DMA Services and EMS respectively). The behavior of EMM386 warrants at least a passing comment. When this memory manager is active, it ensures that the computer is returned to real mode if rebooted by int 19h, Ctrl-Alt-Del, or an explicit jump to the BIOS's initialization point. As part of this, it also restores the first 80h interrupt vectors, with the welcome side effect of safeguarding the system from subsequent programs that neglect their responsibilities concerning int 19h (see "Int 19h and EMM386").

Int 19h and EMM386

Curiously, EMM386 version 4.20 (distributed with DOS 5) does not restore interrupt vectors to the addresses they had when EMM386 was loaded, but rather to their addresses at the time of the memory manager's most recent activation. Consequently, this version of EMM386 is liable to restore the interrupt vectors to unsuitable values if it has been turned off and on again sometime during the session.

Now, it may seem that this problem arises only through contrivance, especially since, in practice, it is often impossible to turn off the memory manager from the command line

```
C:\DOS>emm386 off
Unable to de-activate EMM386 as UMBs are being provided
and/or EMS is being used.
```

let alone toggle its condition. There is a common situation, however, in which EMM386 may be deactivated despite the provision of upper memory and without explicit direction at user level. The occasion is to give way to other protected-mode software (which would otherwise be unable to initialize). Specifically, Windows enhanced mode expects any pre-existing memory manager to respond to the Windows initialization broadcast (int 2Fh function 1605h, discussed later in this chapter) by providing the address of a routine which Windows may call to disable the memory manager (and return the CPU from virtual-8086 mode to real mode). The bizarre consequence, for the present discussion, is that the simple act of running Windows 3.0 destroys EMM386's ability to cover for software that destabilizes int 19h. This has been remedied for EMM386 version 4.44, which accompanies Windows 3.1.

Incidentally, it is unsafe to issue int 19h while EMM386 is turned off (in configurations where this is possible). In this situation, there is much less cleaning up to do, but there remains the duty to restore interrupt vectors on receiving the warm boot request; EMM386 resets the int 67h vector to zero, but apparently forgets that it has also hooked int 4Bh.

Protection for int 19h is also increased as a side effect of configuring a stack pool for hardware interrupts. For instance, KEYB installs an int 09h handler but does not also hook int 19h; this oversight is covered if stack-sharing is enabled, but int 19h may crash the system if KEYB is loaded and **stacks=0,0**. A small "damned if you, damned if you don't" irony follows the observation that stack-sharing is configured only after device drivers are loaded. The stack pool serves its intended purpose with respect to any given hardware interrupt only if it is the last to hook the interrupt, but int 19h must be protected by whichever program hooks the interrupt first. To gain the safety of stack-sharing for the mouse interrupt, it is therefore necessary to use the MOUSE.SYS version of the Microsoft Mouse driver, but this means that int 19h might not succeed in restarting the system. In contrast, when the MOUSE.COM version is installed, stack-sharing is defeated for the mouse interrupt but serves instead to cover the mouse driver's failure to protect int 19h.

Irony is especially rich in the case of the SMARTDRV.EXE disk cache distributed with Windows 3.1. This program hooks int 19h to make sure that dirty cache elements are written to disk, but neglects the restoration of interrupts. Among the vectors concerned are interrupts 08h and 09h, so unless some other program (or the stack-sharing module) will take care of these vectors after the disk cache has dealt with int 19h, SMARTDRV.EXE achieves the reassuring effect of saving cached data to disk and then crashing (rather than warm booting) the computer.

Now, there cannot be many programs that attempt to reboot via int 19h, so the loss of the facility might appear unimportant. A good many texts and articles about TSR programming make no mention of obligations with respect to int 19h, and, possibly in consequence, many readers are likely to regard the topic as obscure. The attitude adopted may come down to one's picture of the way the world works: Poorly considered advice on resident programming has converted a documented BIOS interface into a system instability, but do standards of safety really matter if the market is unharmed or unconcerned?

For instance, it might not matter to anyone that Windows enhanced mode—an apparently sophisticated, protected mode operating environment—falls apart if int 19h is executed (see "Int 19h and Exiting Windows"). Perhaps there is a deliberate policy (among developers but not yet communicated to BIOS manufacturers) that the existence of int 19h should be ignored; more likely, it is just that a lot of programmers and writers have never directed their attention to the matter. Deliberate policy is preferable because at least it is not incompatible with claims of expert skills and extensive testing. Anyone who thinks the matter ignorable may care to consider their reaction to losing precious disk data because a write-behind disk cache had not trapped int 19h to flush cached data to disk.

The last word on int 19h might be left with IO.SYS itself. Observe that two vectors join Table 2-1 only with the release of DOS 5. Of these, int 10h is not actually hooked by

IO.SYS, but its address is recorded at startup and restored before int 19h is allowed to reach the BIOS. Whether or not this is a remnant of some other intention may never be known, but the consequence is that DOS 5 safeguards int 19h against installation of ANSI.SYS or DISPLAY.SYS (neither of which traps int 19h when hooking the BIOS interrupt for video services). Perhaps the best approach in future would be to have IO.SYS save all interrupt vectors during its initialization; it could then restore them all in the event of a warm boot, using the mechanism it possesses already and at a cost to memory of just 1K (which would hardly be noticed in the HMA).

Int 19h and Exiting Windows

That Windows compromises the safety of int 19h is all the more remarkable given that in standard mode, Windows 3.1 uses int 19h as its method for restarting the system (say, because the user elects to have the Setup program's changes to CONFIG.SYS and AUTOEXEC.BAT take immediate effect).

The several possible (orderly) ways to leave Windows are specified through the exit code passed from one Windows component to another as Windows unloads. A typical sequence starts when a Windows application calls the `ExitWindows` function, whose first argument is ultimately the exit code returned by WIN.COM to its DOS-based parent. In both versions 3.0 and 3.1, WIN.COM interprets the exit code 42h as a request not to exit to DOS as usual, but rather to restart Windows immediately. In version 3.1, WIN.COM understands the exit code 44h as a variation in which it restarts Windows after running another program. The pathname and command line for this program are taken from a buffer (80h bytes each) whose address the `ExitWindowsExec` function discovers by calling int 2Fh function 4B20h (defined by WIN.COM to clear ax to zero and point dx:cx to the buffer).

The case on which present interest centers—restarting the system—is filtered off at higher levels still. In Windows 3.0, the Setup program implements its own system restart, as discussed briefly in "Ctrl-Alt-Del and Windows" later in this chapter. Only Windows 3.1 unifies its treatment with the exit code scheme: The `ExitKernel` function exported by KRNL286 and KRNL386 (and serving as the next level down from `ExitWindows` in USER) understands the exit code 43h as a signal to restart the system. In Windows enhanced mode, the 386-specific version of the kernel asks the REBOOT virtual device to take over, whereupon the matter is treated as if Ctrl-Alt-Del had been pressed without the benefit of the **LocalReboot** feature. In standard mode, both versions of the kernel attempt to restart the computer by issuing int 19h.

Besides Interrupt Vectors

Unfortunately, the need to restore conditions for the new DOS session does not stop at the Interrupt Vector Table; there are many other changes that may persist across an int 19h reboot to the possible detriment of the new DOS session. The most obvious are changes to data recorded by the BIOS during its initialization and accepted without question by the newly reloaded DOS.

An important example (which will recur) is the word at 0040:0013, whose value during IO.SYS initialization is accepted as the number of kilobytes of RAM fitted to the computer from address zero upward. Ordinarily, the BIOS determines the amount by testing for the RAM's physical presence and the value is an accurate measure of conventional memory. Some amusing and instructive effects may be created by using int 19h to reboot DOS after reducing this BIOS variable. Thus,

```
A:\>symdeb
-ew 0040:0013 0200
-a
0FFE:0100 int 19
0FFE:0102
-g
```

reboots DOS into a world where only 512K of conventional memory seems present. In this configuration, DOS manages memory only up to (but not including) segment 8000h. Memory is physically accessible beyond that segment, but, quite reasonably, DOS does not attempt to repeat an analysis for which the BIOS is much better placed—any more than DOS would (or should) question the BIOS's reports of disk-drive characteristics. The result is as good a reproduction as can possibly be made of a machine that genuinely has only 512K of RAM in the first megabyte.

The instructive aspect is to see how a memory manager may supply RAM for the 128K that is now "missing" from the conventional memory area. Edit CONFIG.SYS to contain a line similar to

```
device c:\windows\emm386.exe noems ram=8000-9fff
```

perhaps with additional parameters to create upper memory blocks or define an EMS page frame. Reboot, again via int 19h—and without changing the word at 0040:0013 from its value 0200h. DOS restarts under the impression that the conventional memory area contains just 512K, but when MEM is run in the new session, it reports the presence of a full 640K.

It is not that EMM386 has rediscovered the missing RAM, but rather that it has built a virtual machine in which segment addresses from 8000h to 9FFFh refer indirectly (through the 80386 page tables) to memory outside the first megabyte. MEM sees no difference between this situation and a "genuine" 640K of conventional memory because

EMM386 realigned the DOS memory arena headers when it realized that new memory was being appended (in the linear address space of the virtual-8086 task) to the original physical memory. To keep good order, the word at 0040:0013 will also have been revised upwards (from 0200h to 0280h), as may be seen by inspecting its value with a debugger. This incorporation of extra memory into an initially deficient conventional memory area is known as "back-filling" (see "Back-filled Memory").

Resume the demonstration by removing the range **ram=8000-9FFF** from the EMM386 command line and rebooting yet again. Remember that the session started with the value 0200h for the word at 0040:0013, which was then increased to 0280h only as part of EMM386's back-filling operation. EMM386 is solely responsible for having changed the BIOS's memory count and ought, therefore, to restore its value to 0200h when cleaning up for the reboot. That EMM386 neglects this step presents no real problem in the contrived circumstance created here, but had the back-filling been truly necessary (for a machine fitted with only 512K in the first megabyte or configured that way through chipset-specific hardware), then rebooting with int 19h would be unsuccessful, since the newly-loaded DOS would believe 640K to be physically present.

This effect, along with the other int 19h problems described so far, is perhaps only a small problem which many will dismiss as "subtle." In the case at hand, this would be especially easy to do, because if there ever was a fashion for 80386 machines with just 512K in the first megabyte, it must surely have passed. I would not dispute a deliberate design decision that sets a program's domain of validity according to hardware compatibility, but inasmuch as this book sets out to probe sensitive boundaries (within DOS specifically and the practices of DOS programmers more generally), it must be asked whether programming oversights are not too often excused by claiming a deliberate support restriction and, failing this, by appealing to "subtlety." The latter suggests that information available to the programmer during development was poor in quality or somehow incomplete and to assert that the solution lay beyond reasonable expectations. Advanced often enough, however, these explanations translate into questions about the average depth of programming talent and about the overall guidance provided by books and articles on PC programming.

There is enough controversy just in noting these questions without also progressing to answers (which the author, who has not worked as a professional programmer, is arguably not entitled to attempt). The main point is that "subtlety" is very much a relative quality whose value in the case at hand is surely diminished by the observation that this need to prevent changes to the BIOS configuration variables from propagating between DOS sessions is well-known; the only difference is the context. It has been noted already that the multiple DOS sessions run as virtual machines by Windows enhanced mode share much of their respective address spaces and that sensitive data in global memory is instanced (maintained separately for each VM) in order to isolate each DOS session from changes made in others. The same principle applies to the (admittedly much less flexible) transition between DOS sessions via int 19h. The instancing of BIOS configuration variables is shown quite clearly in the source code for the comm, keyboard, and printer

VxDs provided with the Windows Device Driver Kit. That the amount of conventional memory is subject to instancing is evident in the way that the linear address space for each VM may be assigned a different amount of conventional memory. The two strands come together in the DOSMGR device, which accepts responsibility for maintaining each DOS session's possibly different value for the word at 0040:0013, just as it also instances the Interrupt Vector Table.

Back-filled Memory

It may seem an obscure concern, but memory managers faced with machines that have less than their full 640K of conventional memory go to some trouble to perform this mimicry (back-filling). The main consideration is that many programs do not seek additional memory outside their initial allocation. If the memory manager does not adjust the DOS memory arena chain and revise the BIOS memory record, then at best, the extra RAM it builds into its virtual DOS session may benefit only those programs which ask explicitly for additional allocations (and are also aware of any special memory strategies involved). With back-filling, a program's first block will be as large as if 640K had been physically available.

Memory managers differ substantially in their attitudes to back-filling. EMM386 must be told explicitly to provide the extra RAM below segment A000h—which it then incorporates into conventional memory automatically. For some other memory managers, the default behavior covers both discovery and mimicry. For instance, Quarterdeck's well-regarded memory manager QEMM386.SYS requires the keyword **nofill** to prevent automatic back-filling and interprets the phrase **ram=8000-9FFF** as a specific direction to treat the extra 128K as upper memory rather than fuse it with the existing conventional memory.

Windows Enhanced Mode

Another program which uses the 80386 paging machinery to create a non-physical 8086 address space is Windows enhanced mode. Of course, Windows has much greater ambitions than a typical memory manager. The latter creates just one virtual machine to overcome limitations in the physical memory map below 1MB. It is essential to the memory manager's purpose that the VM's linear address space should differ from the original 8086 address space. Windows enhanced mode, on the other hand, is a more sophisticated system offering the prospect of running several DOS sessions more or less independently of one another and with the emphasis shifted to pre-emptive multi-tasking. Conceivably, Windows could meet this different goal by creating each VM's linear address space as an exact replica of the 8086 address space that existed before Windows was started.

For historical reasons however, Windows is not so extreme. Windows can back-fill. Moreover, Windows 3.0 back-fills by default, which can run afoul of the DOS 5 **loadhigh** mechanism. Windows 3.1 is more restrictive. For instance, it builds EMS mappable pages into its VMs if the pre-existing DOS session has no EMS driver (though this can be disabled by placing **NoEMMDriver=on** in SYSTEM.INI). The trend, however, is to assume that memory managers outside Windows have already set up the DOS session the way the user wants it and that Windows' job is merely to clone the session over and over. Thus, Windows does not build upper memory into its VMs at addresses where the pre-existing memory manager had not.

By default, when creating a VM in Windows 3.0 the WSHELL VxD asks for (though does not require) a full 640K of conventional memory. In Windows 3.0, the V86MMGR VxD maps RAM to 640K unless something prevents it.

Windows 3.1 introduces the concept of a **LastV86Page**. This is determined first during the real mode initialization of WIN386.EXE (which contains WSHELL, V86MMGR, and other built-in VxDs) to be the page below the one containing the segment past the top of the initial WIN386 allocation (the word at offset 02h in the WIN386 PSP), but revised to be the page containing the first memory arena header above the initial WIN386 allocation if this is also the segment given at offset 02h in the WIN386 PSP and has the form XXFFh. On a 640K machine, **LastV86Page** is usually 9Fh. A lower value may be revised during V86MMGR initialization.

Ignoring the possible complication of an EBIOS, the scheme is as follows: If there is no **PerformBackfill** setting in SYSTEM.INI, then in addition to the usual considerations that might prevent mapping RAM up to 640K, require LastV86Page < 90h and LastV86Page = XFh (so it is assumed that memory is added in multiples of 64K). If all this is okay, treat the case just as if **PerformBackfill=on.** It then remains to check with the DOSMGR VxD that back-filling is allowed. DOSMGR gives the okay if there is no memory arena header above the WIN386 block. Otherwise, back-filling is prevented if that memory arena header is at a segment XXFFh in the current LastV86Page or if that arena header is in a page below LastV86Page (which won't usually be possible). Even after all this, the Windows 3.1 V86MMGR needs to be told explicitly to map the RAM through an **EMMInclude=8000-9FFF** or **UseableHighArea=8000-9FFF** (sic), the only difference between the two keywords being their granularities of 16K and 4K, respectively.

As a general principle, Windows must preserve the linear-to-physical mapping of any memory already allocated to software while a pre-existing memory manager was in control, but may do whatever it likes with addresses that refer to free memory or to its own allocation from DOS (the useful contents of which are moved to extended memory as part of WIN386's real-mode initialization and replaced when WIN386 returns to real mode for termination). The strict requirement is of course that all relationships expected by the original memory manager must be restored when Windows is finished.

This transfer of information is fairly complicated in itself and is made no easier by the different levels of upper-memory allocation. The large-scale structure of memory that will be seen by the Windows' virtual machines (including memory not within the 8086 address space but available to VMs through the EMS and XMS interfaces) is built by the V86MMGR device, based on the typically page-granular allocations of the pre-existing memory manager. Attending just to this, however, would mean missing the finer, paragraph-granular detail of UMB suballocation to DOS programs (through the usual int 21h functions, at least under the DOS 5 scheme). This sort of disparity is seen at user level when EMM386 reports that the "total upper memory available" is okay while MEM declares enough free upper memory for a few more TSRs.

From a Windows perspective, the effect manifests itself in the behavior of Windows 3.0 under a **dos=umb** regime in DOS 5. Windows 3.0 confines its attentions (more or less) to the safe acquisition of the pre-existing memory manager's UMB allocations. No attempt is made to reproduce free upper memory (as understood at DOS level outside Windows) in the VMs. Page granularity is shown by the loss of 4K at the top of conventional memory just because DOS 5 keeps a single paragraph aside to support a link to upper memory.

Windows 3.1 at least acknowledges that areas of upper memory which seem to be allocated at memory-manager level may seem free to DOS. If **LocalLoadhigh=on**, then the DOSMGR device arranges for these areas of free upper memory to be reproduced in the Windows VMs—but still with a 4K granularity. The arena header that links the two memory control chains does not carry a penalty of 4K in Windows 3.1 because various components are aware of the DOS 5 mechanisms for UMB support. Even during the real-mode initialization of WIN386.EXE, it is understood that this one paragraph should not prevent the localization of the last 4K page. In the final stages of a virtual machine's execution, the DOSMGR device copies that vital arena header into place.

Memory Already Back-filled Outside Windows

Both Windows 3.0 and 3.1 have some difficulty with memory that is seemingly below the boundary between conventional and upper memory but which is already subject to paging. Where EMM386 back-fills the last 128K of its virtual machine's conventional memory area, EMM386 disallows the initialization of Windows enhanced mode

```
EMM386: Unable to start Enhanced Mode Windows due to base memory back fill.
```

whatever the Windows version. Someone at Microsoft or Compaq (whose CEMM is essentially the same program as EMM386) clearly knows that Windows has a problem here. Some third-party memory managers however, are not as fussy, and although no opinion is ventured as to whether this arises through lack of proper warning by Microsoft or through the third party's failure to recognize potential trouble, the opportunity to relate the consequences cannot be resisted.

Readers may prefer to repeat the earlier experiment (using a debugger to alter the Kbytes value at 0040:0013 and then warm-booting via int 19h) with a third-party memory manager instead of EMM386. To do this, use int 19h to reboot DOS into a configuration where the BIOS reports only 512K of conventional memory and use a memory manager, such as Quarterdeck's QEMM386.SYS, to backfill the missing 128K. Bearing in mind some earlier observations about memory managers being ill-behaved with respect to int 19h, start the experiment from genuine real mode (or from an EMM386 machine, subject to the earlier remarks on Int 19h and EMM386); also, if the memory manager back-fills by default, then **ram=8000-9FFF** will probably not be wanted on the device driver command line.

Suppose in particular that the first DOS session is configured with **dos=noumb**—with a complete 640K of physical memory in the conventional memory area, the last 128K of which is all free when DOS is rebooted. Nothing in the new session will see any of that last 128K. Instead, the corresponding address range is back-filled by the third-party memory manager. Additionally, either of the following is to be arranged before running Windows:

- The new session is set up with **dos=umb**. It doesn't matter whether device drivers and programs are loaded into upper memory using DOS's own **devicehigh** and **loadhigh** mechanisms or the proprietary methods accompanying the third-party memory manager—or indeed, if upper memory is not actually used.
- With Windows 3.1, the new session has a **dos=noumb** configuration in which some of the last 128K is already allocated by the time Windows is started—even a single paragraph immediately below the 640K mark (the EATBLOCK program presented in Chapter 4 helps arrange this). Incidentally, Windows 3.0 is disallowed for this case because even in normal circumstances, the real-mode portion of WIN386.EXE from Windows 3.0 refuses to start (complaining of memory fragmentation) if its initial allocation is not the last block in the memory arena chain.

In both cases, Windows will almost certainly not load:

```
Either memory is insufficient to run KRNL386.EXE or the value of the
Windowmemsize entry in the SYSTEM.INI file is too large.
You need to run the Setup program again.
```

To understand what has happened, consider that when Windows is started in the second DOS session, it sees a conventional memory area of 640K which must be reproduced for the System VM (this is the first VM created by Windows and the one in which the WSHELL device will run the KRNL386.EXE program). Perhaps surprisingly, even though Windows may perform back-filling for its own VMs, it is entirely ignorant of the possibility that the VM it inherits from the pre-existing memory manager may contain back-filled address space already. Thus, the conventional memory area of the System VM's linear address space is reproduced from 640K of *physical* memory, the last 128K of which persists from before the int 19h warm boot! As expected, this results in some confusion—which, of course, this latest variation of the back-filling experiment is designed to maximize (consider that before the warm boot, that last 128K contains no memory arena headers, but that by the time Windows starts in the new session, the memory arena chain passes into and possibly out of this 128K).

Actually, Windows' discomfort can be increased, provided readers are willing to see their computers hang. Suppose that **dos=umb** on both sides of the int 19h warm boot, but that the starting segments for upper memory are different for the two sessions. This can be contrived easily enough by changing the address range on the memory manager command line before rebooting. A more subtle example arises as follows:

- Start the experiment with the Windows 3.1 version of EMM386.EXE in its **noems** configuration. This has the side effect of reserving the first few kilobytes of upper memory for some of EMM386's own image (to reduce the footprint in low memory). Thus, if EMM386 is to provide upper memory according to **ram=C800-EFFF**, the first arena header in upper memory will be at some segment such as C8E4h.
- With int 19h, reboot into a 512K machine. Have a third-party memory manager back-fill the missing 128K and supply UMBs in the same range. Continuing with the particular figures in this example, the result will usually be that the first arena header in upper memory is at segment C801h (say, because the memory manager uses the paragraph at C800h for internal purposes).

When Windows tries to start in the new session, it sees a memory arena chain go to segment 9FFFh. There is no confusion below this address (assuming that no memory is already allocated above the block given to WIN386), so the conventional memory area in the System VM's linear address space will be created correctly. Thus, Windows runs, but in its VMs the memory arena header at segment 9FFFh is not the one from the DOS session that existed immediately before Windows was started; this vital arena header is instead copied from physical memory because Windows assumes that paging does not apply in the conventional memory area (and disregards any contrary indication from a pre-existing memory manager). Returning to the particular figures, this means that if a memory mapping utility is run inside Windows, it will follow the memory arena chain into upper memory—to segment C8E4h (as would have been correct before the warm boot), not segment C801h (where the upper memory chain really begins). A robust memory mapper will detect that the chain seems corrupt. Unfortunately, **mem /d** does not validate prospective arena headers carefully enough (it checks only for the 'Z' signature of the last header) and is liable to hang.

VDISK Headers and the HMA

Starting with DOS 5, the int 19h handler in IO.SYS clears two paragraphs of memory at 1MB if DOS has been using the HMA. As with the Interrupt Vector Table and the BIOS data area, this memory serves some software as a fixed point of reference, but is not cleared when the BIOS reloads the operating system. Therefore, software that inspects the 1MB boundary in the new session may find effects persisting from beforehand unless either all agents of change in the old session intercept int 19h to ensure restoration or DOS itself (in the new session) guarantees to place the HMA in a known state.

The sense in which DOS 5 is concerned about memory at 1MB having special meaning arises from the use of the first two paragraphs as a control block for VDISK memory allocators—of which VDISK.SYS distributed with PC-DOS 3.20 or 3.30 is the archetype. Readers with a copy of VDISK may care for a small experiment if they also have a system disk for a DOS version lower than 5. Boot the computer with the early DOS version and, to eliminate all other factors, use a CONFIG.SYS file containing nothing but **device** directives for VDISK and HIMEM (neither of which cares much about the DOS version):

```
device a:\vdisk.sys /e
device a:\himem.sys
```

A detailed analysis of problems with VDISK is presented in Chapter 10, but for the moment it suffices to know that a method of allocating extended memory by mimicking the original VDISK is highly unsound and that programs such as HIMEM which supply services defined by the Extended Memory Specification (XMS) quite rightly refuse to load if they even suspect that VDISK is present:

```
ERROR: VDISK memory allocator already installed
       XMS Driver not installed
```

Edit the CONFIG.SYS file so that VDISK will not be loaded, then reboot via int 19h. This time around, there is no VDISK driver in the new session, but HIMEM again refuses to install and even cites the same objection—as if VDISK were still there.

The explanation is that HIMEM cannot know that the VDISK signature it finds at the 1MB boundary is a harmless remainder from a previous DOS session. The only available remedy is that DOS must guarantee the removal of any persistent VDISK header before software in a new session tests for VDISK's presence. If only with hindsight, it is clear that this support ought to have been built into the DOS version released immediately after publication of the XMS, but apparently the issue escaped the attention of the industry leaders who drafted the XMS, despite the close link between VDISK's problems and its misuse of int 19h.

By now the nature of the remedy should have acquired a familiar ring: New versions of DOS must trap int 19h to remove any possible VDISK header before allowing warm boots to proceed and must make sure of the obliteration as the new session initializes. The two phases have an equivalent effect if the same DOS version is used on both sides of the reboot, so one of them may seem redundant. They serve distinct purposes, however, because trapping int 19h also supports rebooting from a new DOS version into an old one, whereas removal at startup is needed to catch potential problems when the reboot is from an older DOS version to a newer one.

DOS 5 (specifically, the SYSINIT module) always clears the first two paragraphs of the HMA at startup, which means that the VDISK experiment does not produce its curious result if rebooting into DOS 5. At the time, no XMS services are available to deal with the hardware-dependent aspects of A20 control, so DOS must resort to BIOS interfaces to gain access to the HMA. It happens that int 15h function 87h, which steps up to protected mode temporarily in order to transfer data to or from extended memory on the caller's behalf, also disables the A20 when it returns to real mode. Thus, DOS 5 takes care of a second sense in which memory at 1MB has a special meaning capable of persisting across a warm boot: Software in the new session may expect to find a disabled A20 (as would normally be the case after BIOS initialization).

Curiously, the IO.SYS handler for int 19h (in DOS 5) clears the first two HMA paragraphs only if **dos=high**. In this configuration, DOS itself places a VDISK header at the 1MB boundary to deceive VDISK allocators into thinking that the first 64K of extended memory is already in use. This is an eminently sensible precaution to ensure that system components in the HMA do not get overwritten by software that either uses or tolerates this discredited allocation technique (though the provision is really only necessary when the XMS server has left some amount of memory contiguous to the 1MB boundary, that is, including the HMA, available for use by non-clients). For a detailed study of the interactions of a typical XMS server with VDISK and int 15h allocators of extended memory, refer to Chapter 10.

Providing for int 19h to remove this fake VDISK header when DOS is using the HMA merely ensures that the favored configuration of DOS 5 (**dos=high**) can be rebooted via int 19h into an earlier version. It does nothing for the case in which a VDISK allocator is installed in a **dos=low** configuration of DOS 5 (again, rebooting into an earlier DOS version). The cases covered seem to be just those which would be encountered during routine testing of an operating system upgrade. There is the strong suggestion that even now provision for int 19h in respect of VDISK headers (at least) arises only through the occasional discovery of its necessity rather than through the careful analysis reasonably expected of industry leaders and of those regarded as experts.

Int 15h—Preparing for Ctrl-Alt-Del

Starting with DOS 5, IO.SYS traps int 15h to monitor the keyboard for a Ctrl-Alt-Del sequence. On some machines, the quick BIOS re-initialization triggered by Ctrl-Alt-Del cannot be relied on to clear memory (at least not outside the first megabyte), so although the Interrupt Vector Table and BIOS data areas will be rebuilt, it is possible for a VDISK header to persist across a Ctrl-Alt-Del reset as well as an int 19h warm boot.

In this situation, all the arguments of the preceding section apply again: The VDISK header will certainly be obliterated if the incoming DOS session is version 5, but in addition, DOS should attempt to detect an imminent reset and rid the system of any VDISK remnants before the BIOS restarts. Again, the reality is that DOS deals only with the obvious case of its own making rather than its more general duty: The int 15h handler's role of wiping the first two paragraphs at 1MB is activated only if **dos=high**.

Detection of Ctrl-Alt-Del is implemented through int 15h rather than by intercepting the hardware interrupt. This carries an important implication for programs that attempt to reboot the computer as if Ctrl-Alt-Del had been entered at the keyboard—the sort of thing sometimes attempted by setup programs (particularly when driven through batch files) and the sort of gadget occasionally presented in the problem pages of magazines, often in the form of a DEBUG script designed to prepare instructions similar to

```
mov     ax,0040
mov     es,ax
mov     word ptr es:[0072],1234
jmp     FFFF:0000
```

These are indeed the instructions with which a typical int 09h handler arranges to restart the BIOS in response to Ctrl-Alt-Del. However, some software, especially disk caches, memory managers, and now IO.SYS, may rely on catching the Ctrl-Alt-Del sequence for last-minute actions to perform before a reset—to flush cached data to disk, for instance. The instructions presented here deny such software a chance.

Memory managers running the DOS session as a virtual machine may side-step this problem by arranging the 80386 page tables so that the address FFFF:0000 refers to a BIOS copy somewhere in RAM; then whenever the instruction stream reaches that address, control may be diverted to more substantial code that performs any necessary cleaning up (see "Ctrl-Alt-Del and Windows," later in this chapter). Other software must detect the keystrokes. Moreover, their method ought to be one that a reboot program can stimulate; intercepting the hardware interrupt generated by the keyboard is a sure way to trap Ctrl-Alt-Del, but with all the will in the world, a reboot program would have trouble issuing a credible hardware interrupt.

A middle path exists on many machines—and for all machines when the DOS 5 version of KEYB is installed—because the int 09h routine that services the hardware interrupt issues a software interrupt precisely for the benefit of programs that need to pre-process keystrokes, sparing them from the low-level aspects of keyboard handling (which can vary substantially across different models and keyboard types). This is the facility—int 15h function 4Fh with the keyboard scan code in register AL—through which IO.SYS obtains its notification of keystrokes.

The Ctrl-Alt-Del combination is revealed when register AL contains the scan code 53h representing the Del key and the byte at 0040:0017 has bits 2 and 3 set to show that the Ctrl and Alt keys (respectively) are both held down. Thus, code designed to trigger the Ctrl-Alt-Del style of warm boot may alert interested programs by executing instructions such as

```
mov       ax,0040
mov       es,ax
mov       ax,4F53
cli
or        byte ptr es:[0017],0C
stc
int       15
```

to simulate the keystrokes before jumping to the BIOS initialization. It is arguable that this code should also check the carry flag after int 15h returns. The default behavior of the int 15h service is to set the carry flag. A clear carry flag, however, directs the int 09h handler simply to complete any necessary hardware-dependent operations, but to otherwise regard the keystroke as fully processed (instead of pre-processed). Without meaning to complicate the matter, it should at least be considered that software (such as HyperDisk) may have trapped int 15h with the intention of defeating Ctrl-Alt-Del.

The point should not be lost that the need for simulating the Ctrl-Alt-Del keystrokes via int 15h has existed for a very long time, at least in the eyes of careful programmers. DOS 5 and the increasingly common use of disk caches merely bring the matter to a state where its omission must be regarded as inept. A practical consequence is seen in the following cautionary note taken from a discussion of SMARTDRV.EXE in the Windows 3.1 Resource Kit:

> If you use a third-party program to reboot your machine from a batch file, you should make sure that you have the above line [**smartdrv /c**] in the batch file before the reboot program. Failure to do so may cause loss of data.

The **smartdrv /c** switch writes all write-behind information to the hard disk. When the same SMARTDrive write-behind cache support was added to DOS 6, failure to heed this advice did indeed result in data loss, which was incorrectly interpreted by at least one major trade publication.

SMARTDrive's command-line method for flushing the cache can be used before rebooting, and definitely should be used if you reboot with a batch file containing some third-party program which does the actual reboot. This last practice is common with batch-file installation programs that make changes to CONFIG.SYS and then, very generously, reboot the system for you. Such batch files should be altered to run **smartdrv /c** before the tiny program that does the reboot. The necessity arises because many reboot programs simply set the byte at 0040:0072 to 1234h, then jump to FFFF:0000. Of course, this by-passes the disk cache, hence the point made earlier that such programs should issue int 15h function 4Fh to simulate Ctrl-Alt-Del before jumping.

Consider also how Microsoft's VSAFE (a memory-resident virus monitor) interacts with SMARTDrive's write-behind cache. It is crucial to load VSAFE *before* SMARTDRV. If you do it the other way round, then VSAFE hooks int 09h, sees any Ctrl-Alt-Del before SMARTDRV does, and jumps to the BIOS to stop any boot-sector virus (that might also have trapped int 09h) from interfering. Neither does it issue int 15h function 4Fh (since the virus may have got that too). The problem, of course, is that you lose the data from SMARTDRV's write-behind disk cache! So this feature of VSAFE loses your data and doesn't actually bring any sort of compensating benefit. (The really amazing thing for users outside the United States is that KEYB is just as adequate a "protection" measure, since it replaces the BIOS int 09h handler entirely.)

In addition to its /c switch to flush the write-behind cache (which is implemented with a call to int 2Fh function 4A10h BX = 1; see Chapter 13), SMARTDRV.EXE has always been configurable to disable write-behind caching. The first two versions enabled write-behind caching for fixed local drives with two FATs (to miss RAM drives, which ordinarily have only one FAT), but you could turn this off from the command line after booting or you could configure it in AUTOEXEC.BAT so that write-behind caching was never enabled.

What version SMARTDrive 4.20 brings is the idea of limited write-behind caching, which means that the write-cache will be flushed before you get back to the COMMAND prompt. Note that this really means the COMMAND prompt, not just ordinary program termination. SMARTDRV hooks int 21h function 2522h to catch the top-level command processor resetting its termination address; it doesn't hook int 21h function 4Ch or anything like that. SMARTDRV 4.2 provides an extra int 2Fh function 4A10h subfunction 0008h to turn this limited delay in reappearance of the COMMAND prompt on or off.

Ctrl-Alt-Del and Windows

Curiously, Windows enhanced mode seems to have no defense against jumps to FFFF:0000, which are likely to crash the system because the BIOS tries to re-initialize from an unanticipated state (protected mode). Even more curious is that the danger *is* recognized by the Windows 3.0 Setup program, which will not use this jump to automate a reboot if Windows is running in protected mode or has a (Windows) real-mode EMS configuration. Instead, in these cases, a message box asks the user to perform the necessary manual action, that is, to press Ctrl-Alt-Del.

Taking this particular case of Windows still further, observe that something similar can occur in Windows enhanced mode when a reboot is triggered by actually pressing Ctrl-Alt-Del. In enhanced mode, this key combination is registered with the virtual keyboard device (VKD) to be treated as a "hot key" for the REBOOT VxD. In version 3.1, the latter is able to differentiate prevailing conditions and offer a range of possible actions (if **LocalReboot=off** in SYSTEM.INI), most notably the closure of a single virtual machine in preference to rebooting the whole system. As pleasing as this flexibility may be, it is the reboot option (selected by pressing Ctrl-Alt-Del a second time) which is of interest here.

Once the REBOOT device decides that the computer is actually to be rebooted, it notifies all the virtual devices. As the default, VKD ends up resetting the processor by writing directly to the keyboard controller (since on many machines, this highly configurable unit has a line to the CPU reset pin). There is no argument that when this method is available, it wipes the slate clean—at least as far as the CPU is concerned. However, if **KybdReboot=off** in SYSTEM.INI, then VKD does not step in; REBOOT is left to ask for the user's manual co-operation, just like the Setup program.

Nothing as fancy as a message box is used; instead, a fatal error is generated in order to unload, get back to real mode, write a message in teletype style to a clear screen and sit in a loop, waiting for the user to press Ctrl-Alt-Del (a third time). All this is necessary to accommodate real-mode processes (loaded before Windows) that have an interest in monitoring these keystrokes. Unfortunately, this otherwise considerate shutdown of Windows omits (deliberately, it seems) the usual exit broadcast (int 2Fh function 1606h), leaving programs with a false impression of Windows' status and possibly inducing them to cleanup inappropriately.

Actually, the author must confess to not understanding exactly what goes wrong in a particular case that remains unexplained. After rebooting a 386 clone with an AMI BIOS and no unusual hardware from within Windows enhanced mode and with the keyboard rebooting facility disabled, the EMM386 version 4.44 distributed with Windows 3.1 (but not version 4.20 which accompanies DOS 5) crashes during its initialization in the new DOS session unless it is configured to provide EMS with no mappable pages in conventional memory (say, by specifying **b=a000** on the device driver command line). When deductive logic is beyond one's means, the next best thing is to investigate intrusively. It is easy enough (especially with the techniques of Chapter 6) to write a small TSR to load before Windows which will consume the final Ctrl-Alt-Del, break out of the fatal loop, issue the Windows exit broadcast, and terminate WIN386. On recovering the DOS prompt and even after running a few programs, pressing Ctrl-Alt-Del still causes EMM386 to crash in the new session, unless one of the intervening programs is Windows—proving conclusively that for the machine in question, EMM386 suffers from a lack of proper cleanup by Windows during fatal exits (from the virtual device layer).

On some (presumably few) systems, the keyboard reboot is not available or is unreliable, and the Windows Resource Kit advises that the facility be disabled if the computer hangs while rebooting. This discussion shows that setting **KybdReboot=off** might cause trouble of another sort—the main difference being that in this case prevention lies at least partly within Windows' control. Pessimists, cynics, and alarmists (the number of whom are imagined to be growing as this book is read) are directed to the sidebar titled "Ctrl-Alt-Del, Windows, and SMARTDRV.EXE—All Together Now."

Ctrl-Alt-Del and SMARTDRV.EXE

The SMARTDRV.EXE disk cache traps both int 09h and int 15h in order to commit write-behind data to disk before the computer is rebooted with Ctrl-Alt-Del. In the present context, the program demonstrates the awkward problems posed by direct interference with the main keyboard servicing routine.

As a rule, when int 09h is received, it should be possible simply to read the keyboard controller's data port to obtain a scan code describing the keystroke. This may be done several times for any one keystroke, typically by successive int 09h handlers in a chain. When the keystroke is deemed to have been processed completely, it may be necessary to clear the keyboard or perform other hardware-specific operations. Certainly, an End Of Interrupt (EOI) signal must be sent to the Programmable Interrupt Controller (PIC); without it, an STI instruction telling the CPU to observe hardware interrupts is somewhat ineffective, since the PIC will only pass timer interrupts to the CPU (the only ones with higher priority). DOS 5, incidentally, fixes this long-standing problem in the KEYB utility's int 09h handler, which now sends the EOI immediately after the return of int 15h function 4Fh.

SMARTDRV's difficulty is that when its int 09h handler detects Ctrl-Alt-Del, the keyboard has to be released and hardware interrupts enabled before any disk operations can be started. An easy way to have the low-level work performed without having to deal with hardware variations is to call the previous int 09h handler, but in the case of Ctrl-Alt-Del, such a call will not return. Instead, SMARTDRV clears the bit that represents Ctrl in the status key flags at 0040:0017 and then calls the original int 09h handler. When the presumably harmless Alt-Del has been processed, SMARTDRV can flush the cache.

Readers with a TSR whose hotkey is configurable may care for a small amusement at the expense of SMARTDrive. Load the TSR first, specifying Alt-Del as the hotkey. Install SMARTDRV.EXE with caching enabled for a floppy disk drive—

an inessential step which simply increases the chance of conflict to a near certainty. Now start copying a very large file to the floppy disk, such as,

```
copy c:\windows\system\win386.exe a:\
```

and press Ctrl-Alt-Del just after the DOS prompt returns (so that it looks as if the copy is complete, when in fact much of the data remains in the write-behind cache). Before SMARTDRV displays a message about waiting for shutdown, the TSR with Alt-Del as the hotkey should pop up!

Ctrl-Alt-Del, Windows, and SMARTDRV.EXE—All Together Now

The Windows Resource Kit conspicuously fails to mention a potential cause of data loss while a disk cache has material still in memory—Windows itself. By default, Windows enhanced mode reboots the processor by directing the keyboard controller to toggle the CPU reset pin. When the facility is available, it has the advantage of ensuring that the processor resumes execution from scratch, but equally it brings the current computing session to a very abrupt end. During the earlier discussion of "Ctrl-Alt-Del and Windows," sharp-eyed readers will have wondered already whether having VKD reset the CPU does not pre-empt the interests of other virtual devices or of real-mode software loaded before Windows was started. Indeed, much of the discussion about providing an opportunity for last-minute cleanup before re-initializing the BIOS applies to this situation also; however, Windows does not oblige.

Thus, with SMARTDRV.EXE installed (such that it will still be active when Windows is running), enable the write-behind cache for a floppy disk drive; again, this is not necessary, but serves to make a near certainty of the data loss. With **KybdReboot=on** in SYSTEM.INI (the default), start Windows, go to a DOS prompt, and copy a large file to a floppy disk in the cached drive. When the **copy** command terminates, press Ctrl-Alt-Del twice (once suffices in Windows 3.0 or if **LocalReboot=off**). Provided that the file and cache are large enough, data bound for the floppy disk but still in memory will be lost.

Perhaps the risk turns out to be negligible when caching is applied more sensibly, but users are entitled to be properly informed—all the more so when the Windows Resource Kit speaks reassuringly (and correctly) of how "SMARTDrive goes to great lengths to avoid data loss" (only to have its efforts ruined by the very package with which it is distributed).

Int 13h—Disk Support

By far the most substantial of the IO.SYS interrupt handlers is the support code for disk services. IO.SYS hooks int 13h partly to help the default block device driver with certain bookkeeping tasks, but mainly to correct a few awkward error conditions for the benefit of all programs that use int 13h for disk access (directly or through DOS).

On receiving int 13h, the IO.SYS handler performs two tasks before calling the BIOS:

If the function is a request to format a disk and IO.SYS has installed its support for drives that report change-line errors, then all units that access the physical disk drive involved in the format request are marked as having had a media change. When the default block device driver next attempts to use that drive, it may therefore know that its picture of the drive's contents must be rebuilt.

Before allowing any access to an electronically-compatible 3½" floppy disk drive, the appropriate byte at 0040:0090 or 0040:0091 is set to 93h, which value causes the BIOS to behave as if it were operating a low-capacity 5¼" drive. This feature was introduced with DOS 4. It may be configured for a particular drive by supplying the **/i** switch to a **drivparm** statement in CONFIG.SYS.

For everything but its **/i** switch, the **drivparm** statement is just a user-level front-end to the Set Device Parameters service which DOS provides via Generic IOCTL (int 21h function 440Dh minor code 40h). This means that drive numbers are interpreted at DOS level (using a 0=A:, 1=B:, etc. scheme to relate the value of the **/d** switch to the more familiar notion of drive letter). Since electronically-compatible 3½" floppy disk drives are supported directly at int 13h level rather than through a DOS function, it falls to SYSINIT to deduce which physical drive is indicated by the user's DOS-level specification. It happens that nothing of this sort is considered, nor is any attempt made to reject a **/i** switch used for drives beyond the first two floppy disk drives (the only ones for which the BIOS maintains diskette status bytes at 0040:0090).

When the BIOS handler returns, some post-processing may be necessary, even if the BIOS dealt with the function successfully. It seems that for certain PS/2 machines with model byte FAh, the BIOS implementation of int 13h functions 08h and 15h must be followed by asking the BIOS to also execute int 13h function 01h (ignoring the value that this extra call returns in ax). Until DOS 4, this seemingly spurious step was taken for all models.

Otherwise, further action is needed only if the BIOS returns one of three types of error:

- DMA transfers are restricted to separate 64K pages, measured from address zero. Attempts to transfer data across a page boundary fail, but can usually be corrected by decomposing the request into smaller pieces. Given that sector boundaries inside the request will rarely coincide with the 64K page boundaries, one of the pieces is almost certain to cross a boundary, so IO.SYS must often employ a temporary buffer in its own data segment; only 512 bytes are set aside for this purpose, which should be remembered by software using int 13h to work with formats that have larger sectors than usual. IO.SYS deals only with the standard read, write, verify, and format functions.

- The BIOS reports ECC data corrections as errors. The BIOS does not proceed to the remaining sectors and apparently does not indicate the number of sectors read successfully before the request was abandoned. In principle, the problem is remediable by reading sectors individually. The IO.SYS int 13h handler regards corrected data as safe: int 13h users receive no error code 11h indicating that some of their data required correction by hardware. DOS 4 introduced a fairly broad test for machine identification; DOS 5 attempts correction for all machines but acts only for the read function.
- The IO.SYS handler for int 13h also notes change-line errors (provided that at least one floppy disk drive is capable of reporting them). These errors indicate that the drive door has been opened since the previous int 13h call to the drive in question. Of course, the default device driver routines detect this condition when it is returned from their own int 13h calls, but by watching for the error as part of its interrupt handling, IO.SYS may keep fully informed even of disk changes reported when other software uses int 13h.

All but a tiny portion of this int 13h handler executes in the HMA if **dos=high**. A stub in low memory checks the A20 state, as usual by comparing the paragraphs at 0000:0080 and FFFF:0090. If the A20 is not enabled, then clearly the situation must be corrected before attempting to access the HMA. As with the device driver interface in this case, IO.SYS does not restore the A20 to its disabled state when the time comes to return to the int 13h caller. Thus, yet another interface has the (undocumented) side effect of enabling the A20 when **dos=high**.

An arguably subtle effect not present in the device driver case is that whenever the IO.SYS int 13h handler needs to call the BIOS int 13h routines, it does not simply call them from the HMA. Instead, it returns to another low memory stub to make the external call lest the A20 become disabled. This is ordinarily something that IO.SYS does not worry about; after all, the user's choice of a **dos=high** configuration implies good behavior with respect to the A20. Thus, IO.SYS code executing in the HMA (typically for the default device drivers) issues all other BIOS interrupts without a care for the possibility that they may disable the A20. Code for the block device driver uses int 13h freely, but this is only because it knows that any problem with the int 13h implementation at BIOS level has been dealt with by the IO.SYS int 13h handler (in the same way, for instance, that it need not trouble over DMA page boundary errors).

The well-known exception to the good behavior principle is that each process starts with a disabled A20 until it uses DOS interfaces. The reason (taken up in Chapters 5 and

9) is that a large class of programs have startup code that relies on wrap-around at the 1MB boundary and would not run properly (or would be dangerous to execute) under a **dos=high** regime if DOS did not make special allowance. Surely, though, IO.SYS does not return to low memory when calling the BIOS int 13h services just for some similarly special accommodation of a BIOS implementation that disables the A20 as a side-effect?

The answer may lie in the possible manipulation of the int 13h chain underneath IO.SYS, be this to insert code which acts between the IO.SYS handler and the BIOS or perhaps even to replace the BIOS int 13h routines completely. The relevant interface (int 2Fh function 13h) is discussed shortly; it has not been documented until recently (and then only in a more or less uncommented appearance in the WDCTRL\WDRM-INIT.ASM source file supplied with the Windows 3.1 Device Driver Kit), but has long been a favorite of programs seeking to affect disk access at the lowest level. One plausible motivation for the IO.SYS int 13h handler to defend itself against the A20 being disabled by the lower-level (ostensibly BIOS) int 13h routines is that Microsoft knows of a disk caching program which hooks int 13h underneath IO.SYS and transfers data to and from its cache in extended memory via int 15h function 87h (which disables the A20 unless corrected by software such as an XMS server).

Machine-Specific Int 13h Handling

Even though some aspects of the int 13h handling described here have acquired a machine dependency in recent DOS versions, they were all intended originally to have general applicability. In one or two highly machine-specific situations, IO.SYS itself may install a second int 13h handler, acting underneath the primary one to correct known BIOS problems invisibly to all other interested parties from the main IO.SYS disk handler upward:

- On PC-AT machines (model byte FCh) with BIOS date "01/10/84," IO.SYS replaces some of the BIOS code responsible for reading (functions 02h and 0Ah) from fixed disks. Particular BIOS routines needed during the processing but not requiring alteration are called at known addresses in segment F000h (with the aid of a RETF instruction in the same segment). This suggests that the fault lies with the deepest code replaced in RAM, that is, with the procedure that actually reads data through the drive controller's ports.

- A second case is new to DOS 5 and affects Compaq machines with BIOSes dated earlier than "08/04/86." For fixed disk functions higher than 15h, the handler exchanges the caller's value of register DS for segment 0040h before letting the BIOS process the function. Presumably, Compaq's early BIOS implementations cannot be relied on to load the BIOS data segment and require this correction to work properly.

Int 2Fh—Multiplex Interfaces to IO.SYS

Until DOS 5, IO.SYS offered only two int 2Fh functions. One of these, function 08h, provides access to internal tables maintained for each unit supported by the default block device driver and is the means by which DRIVER.SYS installs external disk drives. It is described in Chapter 16, together with a practical example of its use to discover the int 13h drive number corresponding to a given DOS drive letter.

Managing Int 13h

The other long-standing int 2Fh service provided by IO.SYS is function 13h—an aptly chosen number, given that the function provides the means to manipulate the int 13h chain underneath the IO.SYS handler. This facility appears to be very badly understood, which may be reason enough to examine it here. More than this, however, the function has acquired new significance with the release of Windows 3.1 and its FastDisk feature.

A simple expression of the idea behind FastDisk is that it allows Windows to handle fixed disk access at hardware level in protected mode. The prime motivation (at least from a DOS perspective) is speed; even when a virtual-8086 task has permission to use a particular I/O port, each I/O instruction takes nearly four times as many clock cycles as it would if executed with privilege level 0. By replacing the int 13h services with its own code, Windows gains a clear performance advantage for large numbers of simultansous I/O instructions.

As for the mechanisms behind FastDisk, present purposes demand only a little less than the following overview. The WDCTRL virtual device supplies the hardware-dependent code which communicates with the disk controller. An int 13h entry point is installed by the BLOCKDEV VxD, which also accepts direct requests from other virtual devices, allowing them to avoid the usual constraints on re-entering int 13h; tighter definition of critical sections is another benefit that Windows gains (for itself and for FastDisk clients) by implementing its own code for disk access. The native BLOCKDEV command structure (accessed directly by FastDisk clients) supports serialization of disk access, arranging a queue of pending requests in terms of priority and likely access times. Int 13h requests intercepted by BLOCKDEV are translated into this scheme by passing them to another virtual device, called simply INT13. Only one virtual device supplied with the retail Windows 3.1 package acts as a direct FastDisk client; if possible, the PAGEFILE device uses the BLOCKDEV interface for all access to the permanent swap file, enabling faster (and more flexible) response from the virtual memory paging machinery.

It is important that BLOCKDEV does not simply hook the int 13h vector, since this would disable services (such as disk caching) provided by programs that trapped the interrupt before Windows was started. As far as disk usage by DOS applications is concerned, the intention is for Windows to replace only the hardware-level aspects. The provision of a way to intercept int 13h activity *underneath* IO.SYS is therefore crucial to Windows' scheme for improving access times.

Table 2-2 shows the input and output of the undocumented int 2Fh function 13h service that IO.SYS provides to hook int 13h at a low level.

Table 2-2: Int 2Fh Function 13h

INPUT:

AH	13h
DS:DX	address of int 13h routine to place immediately below the primary IO.SYS handler
ES:BX	address to which IO.SYS should restore int 13h on receipt of int 19h

OUTPUT:

ds:dx	address of int 13h routine that lay immediately below the primary IO.SYS handler before this function call
es:bx	address to which IO.SYS would have restored int 13h on receipt of int 19h before this function call

The function is implemented by exchanging the new addresses with values held in two far pointers. If IO.SYS installed a machine-specific int 13h routine, as described in the previous section, then the "es:bx pointer" (for want of a better name) also serves for the indirect transfer of control from this routine to the next handler.

In DOS 5, interrupts are certain to be enabled when the exchange of addresses takes place (in earlier versions, the situation is merely possible rather than certain). Unless software configures the Programmable Interrupt Controller to ignore hardware interrupts for all the time taken to follow the int 2Fh chain and return, the possibility exists that int 13h services attempted during an interruption may find that one pointer or the other holds a new offset but old segment (and therefore will be routed to invalid code, most likely crashing the computer).

From at least DOS 3.30 onward, the ds:dx pointer lies at 0070:00B4. In DOS 5, a sequence of other data (device driver headers) is interrupted to keep the pointer at this address, as if Microsoft had chosen to make good the idiocy behind some program that accesses the pointer directly (in preference to using the interface).

Observe the unusual design; new int 13h code is installed before the address of the existing handler has been returned, suggesting that the latter is required only for eventual restoration, not for the proper working of the new routines. From the outset then, it seems the intention was to support BIOS replacement, rather than a low-level insertion into the interrupt chain. Incidentally, the term "BIOS" is used loosely throughout this discussion to include any additional int 13h handling provided by ROM extensions (initialized before software, particularly IO.SYS, is loaded from disk).

Applications for Int 2Fh Function 13h

Now, it is easy to imagine this interface's attraction to those whose motives are less than pure, but there are several good (and even compelling) reasons for using int 2Fh function 13h to thread disk-servicing code into the int 13h chain at the lowest level without meaning to disconnect the BIOS. With this larger goal in mind, the interface looks not only primitive but heavily overloaded, and it will be seen before long that the adaptation raises serious technical difficulties, some of which may have no truly satisfactory solution.

BIOS Repair

An immediately obvious use for int 2Fh function 13h is to generalize the provisions that IO.SYS already makes for correcting deficient BIOS int 13h implementations. The generic IO.SYS caters to particular IBM and Compaq machines, and, although other manufacturers could presumably engineer their own IO.SYS version to deal with problems discovered in their computers, they can also distribute a simple TSR that uses int 2Fh function 13h to install the "fix."

Large Hard Disks

A more complicated example arises when access to a disk with more than 1024 cylinders requires software support because translation of disk-drive characteristics is not performed in hardware. Instead, a specially-written BIOS offers an extended form of int 13h parameterization to allow more than the usual 10 bits when specifying a cylinder (typically by reinterpreting the two high bits of the otherwise underused register dh). Such extensions have not been adopted on anything approaching a universal scale, so in practice some additional piece of software must intercept all other programs' int 13h requests and allow them to picture the disk as having fewer cylinders but more heads, all the while translating these requests to fit the extended BIOS interface. Since the IO.SYS int 13h handler understands only the standard parameterization when it decomposes requests that cross DMA page boundaries or works around ECC corrections, it falls to the translator either to ensure that the relevant execution paths in IO.SYS can never be taken or (more easily) to perform its work between IO.SYS and the BIOS.

Large Sectors

The IO.SYS code to spare int 13h clients from troubling over DMA page boundary restrictions provides another potential application. Recall that only 512 bytes are set aside for internal buffering while decomposing problematic requests. The default block device driver does in fact impose 512 bytes as the physical sector size for floppy disks when processing normal read/write requests, but larger sectors (1K) are supported at DOS function level through a Generic IOCTL interface, and, of course, software may work with such media by modifying the Diskette Parameter Table and using int 13h directly. That the sectors are larger than the space allowed inside IO.SYS means that requests which cross DMA page boundaries must be decomposed before the IO.SYS int 13h handler tries to resolve the error.

Now, it may often be possible for a program to dodge the problem by passing all its disk requests through a single buffer, prepared especially to lie wholly inside a DMA page. Suppose, however, that a program working with 1K sectors wants to transfer data to or from arbitrary addresses in memory. Given that a typical disk request is small compared to the DMA page, most requests would pass through to the BIOS and back without error. Checking buffer addresses in advance (whether by calling a dedicated procedure whenever disk access is contemplated or by installing some temporary int 13h code so that details are invisible to other modules of the program) therefore incurs a time penalty, the cumulative effect of which may be quite substantial. The consequent design decision is of no concern here, only that the most efficient way to avoid the penalty is to use int 2Fh function 13h to insert error-correcting code underneath the IO.SYS int 13h handler, so that requests need be decomposed only when an error is known to have definitely occurred.

Simulating Disk Change-line Errors

For an application of int 2Fh function 13h not tied so closely to unusual hardware, consider a TSR with a "pop-up" stage, during which floppy disk operations might occur, perhaps because the TSR asks explicitly for a file or more generally, because the TSR is a task switcher. Now, some may contend that expecting task switchers to deal with floppy disk changes is pushing the system too far, but it is amazing what users will try when space on the available hard disks is scarce (and just as amazing how little protection they receive either from the system at run-time or from writers on PC programming who have such low expectations of what is achievable). Of course, the best way to separate access to different disks through the one physical drive is to use different DOS drive letters (as discussed in Chapter 16), but changing disks under the supervision of a task switcher may seem reasonably sane; after all, the essence of task switching is to preserve important elements of the system state so that an interrupted task may resume with (very nearly) the same operating environment it had at the moment of its suspension.

In programming terms, the situation is perhaps unusual in that the switcher must not only restore the system state for the suspended task but may also have to introduce something that was not present when the task was interrupted. To give the task some chance of realizing that the disk it expects was changed for another while the task was suspended, the switcher may want to arrange the simulation of a change-line error when the task next attempts a disk operation. There are all sorts of reasons why the newly-resumed task might not notice the simulated error, but here, in any case, is the role for int 2Fh function 13h: If a fake error code is wanted for the next int 13h call, then it should be injected into the system as if it had come from the BIOS (so it may be seen by *all* interested parties).

Disk Changes

DOS appears to have a long-running difficulty with removable media. Even with a drive that supports disk-change detection in hardware, it seems all too easy to lose a disk change in its percolation from device driver to kernel and ultimately to application level. The basic provision is that once the DOS kernel identifies the physical drive for a proposed file operation, it asks the underlying device driver to perform a Media Check (device driver function 1).

To get acquainted with DOS's handling of removable media, begin with a relatively simple case where SHARE is installed. Run a program that allows execution of DOS commands (typically by shelling out to a new command processor) but which does not close files on removable media before shelling out. For instance, load SYMDEB or CodeView as if to debug some program, then redirect that program's output to a file on a floppy disk and escape to DOS. On reaching the DOS prompt, change disks and run the **dir** command to get the new disk's directory. DOS cannot fail to interpret the disk change as invalid, for surely, the previous disk (on which there is an open file) is still in use. The result is an int 24h exception, which usually makes its way to user level as some message like

```
Invalid disk change reading drive A
Please insert volume NO NAME     serial 3354-0AD3
Abort, Retry, Fail?
```

Such warnings are undoubtedly more useful when disks are given meaningful volume labels; the main point however, is to show that the disk change *does* get trapped. That said, readers may have noticed a common theme running through this book and now suspect that the real point is to show one or more circumstances, probably involving the interaction between DOS and Windows, where the invalid disk change is not trapped at all.

SMARTDRV.EXE

It will take a while to build up to such an extreme situation (indeed, in the final Disk Change experiment that follows, the disk change won't even be recognized, let alone declared invalid). One reason for the delay is that some readers will have encountered a problem already—even with the simple demonstration presented above—and which must now be explained.

The author stubbornly persists in using the SMARTDRV.SYS disk cache distributed with DOS 5, because it is significantly faster in a working environment that consists more of reading than writing and where the only drives to be cached are supported via int 13h. Repeating the previous demonstration with the same floppy disks but with SMARTDRV.EXE configured to perform some sort of caching (read, write, or both) on the floppy disk drive produces an error message in which the volume name is gibberish:

```
Invalid disk change reading drive A
Please insert volume è #   p serial 0070-06F4
Abort, Retry, Fail?
```

in which case, it doesn't seem to matter whether the required disk is labelled meaningfully or not. Now, this is a pretty obvious error and it may have been fixed in versions of SMARTDRV.EXE released before this book appears in print (the error is still present in SMARTDRV 4.20), but it is worth studying a little, especially because SMARTDRV.EXE implements a highly directed interface to advise Windows 3.1 of media changes. Unfortunately, SMARTDRV does not show nearly the same concern for the integrity of the system (DOS) whose device driver calls it intercepts.

When it finishes processing an intercepted device driver command, SMARTDRV must place the results of its operations into the request header it received from the DOS kernel. As with any block device driver that supports the Open, Close, and Removable Media commands, it must be aware that DOS will expect the device driver error code denoting an invalid disk change to be accompanied by an address at which DOS may find the volume label (and serial number) of the "valid" disk. That SMARTDRV neglects this case is shown by the gibberish which appears when DOS eventually raises the int 24h exception. In the particular case shown, the field in the device driver request header where an address should have been entered still contained zeroes, so the reported volume label is an ASCIIZ rendering of the Interrupt Vector Table and the serial number is the entry for int 03h!

Losing Disk Errors in DOS 5

Now consider the system's response to the Abort, Retry, Fail prompt. With many critical errors, an Ignore option is offered. That it is missing here is an indication of DOS's tenacity: Some process needs a particular disk and DOS will repeat the error condition again and again until that disk is restored to the drive. Again, this is a fairly obvious error that may well have been corrected in a new DOS version by the time this book appears in print.

Incidentally, if **dos=high** in DOS 5, then taking the Fail option may lose the error condition. This is not confined to invalid disk changes: All disk errors encountered while reading into the **buffers** system when it uses the HMA are lost.

Disk Changes and Windows Enhanced Mode

With two floppy disks to hand, one of them empty, the other not, start Windows enhanced mode. Run the dreaded EDLIN to create a file on the first floppy disk (or redirect a program's output under a debugger, as before; there is no need this time to shell out to DOS). EDLIN is chosen for the demonstration only because it will keep the file open. Many other programs have this property, particularly large applications that envisage recurrent access to the same files. Among Windows applications, Microsoft Excel is a widely available example (and the situation described here may already be familiar to readers for some reason constrained to work with spreadsheets on floppy disks). Word for Windows, on the other hand, is unsuitable because it (cleverly) keeps files open only if they are not on removable media. EDLIN offers two advantages for present purposes: it is part of the standard DOS package and, more importantly, its small size eases the experiment's repetition.

So, from EDLIN's asterisk prompt, switch to the Windows shell (such as Program Manager) and create a new VM in which to take the directory of a new disk in the same drive. As with the earlier demonstration, the disk change is trapped as invalid. To Windows enhanced mode, the relationship between a drive and its contents is global; processes running in the second VM are bound by the requirements of a process in the first VM.

Some readers attempting this experiment (but who deviate from the instructions in a way that ought not to matter) may find that quite the opposite occurs—that instead of trapping the disk change, DOS seems to think that the first disk is still present. To increase the chance of this happening, repeat the experiment with a newly booted computer, start Windows enhanced mode, and open two virtual machines immediately—instead of creating the second VM later. In the first VM, take the directory of a disk in drive A specifically (opening a file on the disk is irrelevant now, since DOS won't notice any disk change, whether invalid or not). Switch to the second VM, replace the disk with an empty one, and take the directory again.

The difference here is that, because the second VM already exists, the **buffers** system is undisturbed by the switch (assuming Windows has sufficient memory) and should still contain sectors read from the first disk; in the earlier case, these sectors were displaced by ones that DOS needs in order to load the WINOA386 module which supervises the new VM's creation. Obviously, when the default block device driver responds to the DOS kernel's Media Check by saying that the disk has changed, the DOS kernel should discard any sectors it still holds from the drive in question (and if any of these sectors are "dirty," then DOS should raise a critical error asking the user to reinsert the disk to which these sectors must still be written). That the **buffers** system has broken down is revealed when **dir a:** is attempted for the second (empty) floppy disk in the second VM, only to produce a directory listing for the first (non-empty) disk. If the experiment is repeated—from scratch, for reasons that will become clear shortly—but with the precaution of pressing Ctrl-C or Ctrl-Break before reading the directory of the new disk in the second VM, then the **buffers** system is reset (because the COMMAND.COM handler for int 23h calls DOS function 0Dh) and the disk change *is* recognized.

Pressing Ctrl-C at the appropriate moment is actually the least bizarre factor in the problem. Note first that the case where sectors persist from the first disk is not immediately reproducible. Thus, restore the first disk, return to the first VM, exit from EDLIN, and repeat the final steps of the experiment, that is, reusing the same two virtual machines. The disk change is correctly identified as invalid, as it will be on all subsequent occasions. Second, observe that there is no problem whatsoever if any drive other than A: is used for the experiment. This applies even to a logical partner of drive A, as might be created using DRIVER.SYS (and which exists by default as B: on systems with only one physical floppy disk drive). Third, the problem disappears if there has been any sort of DOS-level access to drive A either before Windows was started *or in the second VM before seeking the directory*. Fourth, the disk change is caught correctly if any drive was used to label a disk, again before Windows was started or in the second VM before accessing drive A.

Industry analysts may care to estimate the damage done (particularly to good will and confidence at consumer level) when software exhibits this sort of "mysterious behavior." Imagine the plight of a user who has just saved an Excel spreadsheet onto a disk in drive A, then switches to a DOS Prompt to copy a file to another floppy disk, perhaps because someone has just stepped into the office and asked for yesterday's figures. This discussion has shown that if the user's habit is to keep a DOS box handy while running Windows applications, then there is a good chance of corrupting the second floppy disk, since directory and FAT information may persist from the first floppy. Worse (in the long run, at least) is the inevitable reaction from technical support staff, who will not only be unable to reproduce the problem reliably, but who will believe they have followed the same steps taken by the hapless user; things like having labelled a floppy disk in another drive while outside Windows or inside the target DOS box are simply too obscure even to enter into contemplation as possibly relevant factors. The outcome for the user must be a diminished confidence in computers and the industry in general.

From the programming perspective, it seems fair to say that the question of accommodating media changes has become a subject for work-arounds rather than genuine solutions. Moreover, there is a significant danger that normal working practice is adapted unconsciously to avoid procedures liable to create trouble unpredictably; after all, a computer user gains competence by seeking the most efficient methods, not by engaging techniques that risk system instability and data loss.

The experimental approach to appraising software (implicit in claims that a program has been tested extensively) is founded in the belief that the circumstances of a bug can be described accurately by a user (perhaps trained for the task of testing software) who stumbles on a problem. Armed with a sufficiently good description, a developer should be able to trace the error and then work on its correction. For instance, readers with sufficient knowledge of the DOS kernel will find that the cause of this problem with disk changes under Windows enhanced mode may be identified from the factors presented above. Against this, the author submits that the circumstances of the problem are sufficiently uncorrelated to defeat any practical attempt at isolating them by experiment; they can be given here only because they have been determined as consequences of a programming error known from inspection of the DOS kernel. Readers are encouraged to wonder how much mysterious behavior in operating system software actually stems from simple programming mistakes that remain unidentified because software analysis is treated as a behavioral science.

DOS manages volume labels on two largely separate levels. One form of volume label is really a root directory entry distinguished by its possession of a special ATTR_VOLUME (08h) attribute. With some limitations, it may be created, deleted, and renamed with the same int 21h functions that operate on files. The major disadvantage is that during this operation, neither the DOS kernel nor the underlying device driver can be entirely sure of the disk's identity.

Immediately after the DOS kernel creates or deletes (but curiously, not when it renames) this form of volume label, it calls the responsible device driver to perform a Media Check. It is important to know that the device driver is able to use the new volume label, but under the circumstances, the device driver's response to the Media Check is immaterial and the contents of the buffers are considered to be still accurate. After all, the DOS kernel knows what the device driver cannot—that the disk only appears to have changed. The Media Check service is called mainly as a prelude to forcing the device driver to rebuild its picture of the disk for future reference. Note that there is some chance that the disk really has been changed in the time taken between the DOS kernel's re-labelling and the device driver's processing of the Media Check command.

Among the DOS kernel's internal variables is one in which DOS records the 0-based drive number when it creates or deletes a volume label. This byte (at offset 0AA1h in the DOS kernel) starts off with an inappropriate value.

In principle, it is a zero-based drive number used by the DOS kernel to resolve a difficulty with the Media Check function when changing a volume label. If the kernel realizes that a negative Media Check is only an artifact of DOS being in the process of changing a volume label, it skips the usual consequences of a disk change. The value FFh denotes that no drive is associated with this purpose. Unfortunately, the variable's initial value (static initialization at assembly time) is 00h, denoting drive A! Because the variable is instanced, each new VM under Windows will show this problem if drive A was not used before running Windows (or in the System VM) and no disk has yet been labelled (which would reset the volume change flag to FFh). You actually don't need a task switcher to experimentally reproduce this problem, but it helps by ensuring that DOS buffers for the first disk in drive A are still there when you look at the second disk.

A program which yields deliberately to another (especially by spawning it as a new process, as described earlier) is in control of its suspension and may prepare itself for a disk change that occurs in the meantime. In the task switching case, however, no such opportunity is sure to exist. It is not just a question of backward compatibility with programs that do not know the particular switcher's interface; this would assume that a program could set itself up as a local switcher client and obtain notification of its suspension before it is swapped out of memory and of its resumption after it has been swapped back in. Under Windows enhanced mode, with its genuinely pre-emptive multi-tasking of DOS applications in separate virtual machines, it is simply not possible for a program run in a VM to learn of its suspension unless it has arranged the co-operation of a virtual device driver operating in ring 0. In practical terms, each program interested in obtaining switcher notification while running under Windows enhanced mode must have either a resident stub loaded before Windows starts or a **device** entry in SYSTEM.INI to load a VxD just in case that program is executed in a VM inside Windows. This is clearly an unwieldy situation matched only by Microsoft's obligation to ensure that as few programs as possible need to cover any shortfall in Windows' isolation of DOS properties.

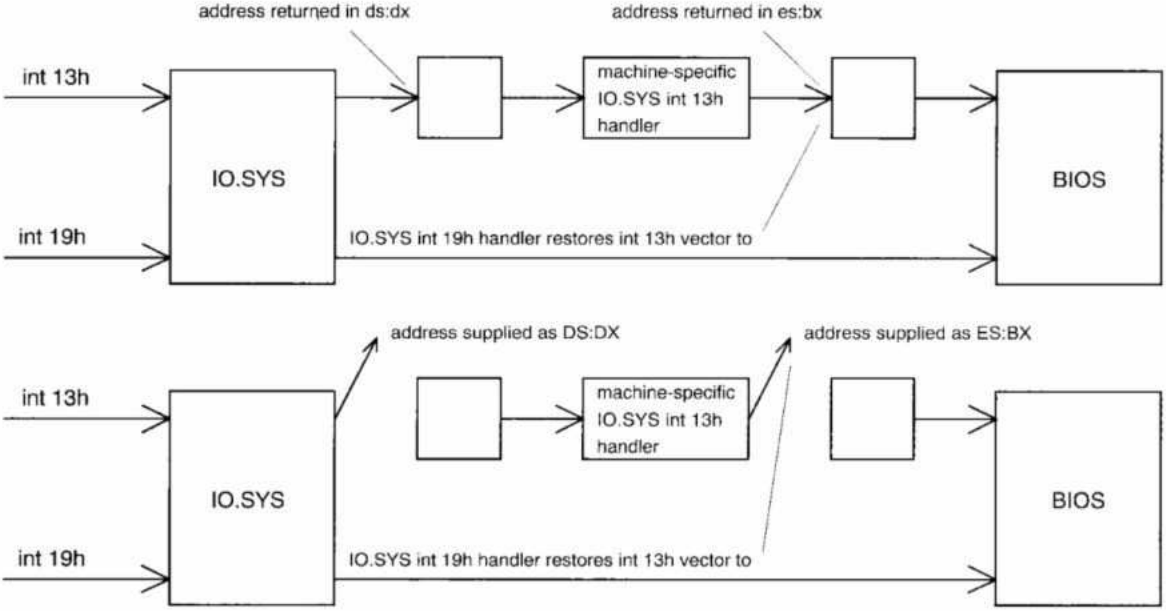
Behind the Int 2Fh Function 13h Interface

It is hard to escape the esoteric nature of the examples given earlier as possible uses for int 2Fh function 13h. Perhaps none of them has ever been implemented in real software, but for all of them, use of int 2Fh function 13h to insert code into the lowest reaches of the int 13h chain is neither necessary or advantageous. Certainly, it must be expected that the interface has been worked beyond the original provision for complete replacement of BIOS disk services (for sensible reasons and before its recent use by Windows). Even in the case of Windows, the FastDisk handler is threaded into the int 13h chain, since the BIOS int 13h routines are still needed for disk drives with no corresponding FastDisk device and for functions that the installed FastDisk devices do not support.

Figure 2-2 depicts the general arrangement of the int 13h chain between IO.SYS and the BIOS (which term, remember, is understood throughout this discussion to also cover ROM extensions and so on that are initialized before software is loaded from disk). Empty boxes in the diagram stand for any number of handlers threaded into the chain by repeated invocation of int 2Fh function 13h. In the special case where the function has not yet been used, the diagram collapses to show that the address of the BIOS int 13h handler is waiting to be returned in es:bx. If IO.SYS has installed a minor int 13h handler to correct BIOS problems, then first use of the function will point ds:dx to this handler, but otherwise, both ds:dx and es:bx will be returned addressing the BIOS entry point for int 13h.

The second part of Figure 2-2 shows how the function affects the int 13h chain. See how the flow of control for int 13h is completely diverted. The BIOS disk services are detached both from the int 13h chain and from restoration before an int 19h warm boot. Except in the rare case where the new ds:dx handler is actually intended to replace the BIOS int 13h services completely, the caller must reconnect the chain after the function returns, typically requiring that the function be used a second time.

Figure 2-2: Manipulation of Int 13h Below IO.SYS


address returned in ds:dx
address returned in es:bx
int 13h
machine-specific IO.SYS int 13h handler
IO.SYS
BIOS
int 19h
IO.SYS int 19h handler restores int 13h vector to
address supplied as DS:DX
address supplied as ES:BX
int 13h
machine-specific IO.SYS int 13h handler
IO.SYS
BIOS
int 19h
IO.SYS int 19h handler restores int 13h vector to

Just to locate the existing int 13h handlers underneath IO.SYS, the function must be called twice—once to obtain the addresses and a second time to restore the chain—as shown in the following program's incomplete *germ* of a C function to return the current es:bx address (on the basis that this will ordinarily be the int 13h handler as close to the original BIOS entry point as can be determined). The program merely displays a message such as "IO.SYS thinks that the BIOS int 13h handle is at F000:930D."

```
/* GETINT13.C */

#include    <intpack.h>     // supplies PUSHA structure, etc., for access to
                             // caller's registers in functions declared with
                             // the _interrupt keyword in Microsoft C -
                             // may have to be replaced for other compilers
#include    <standard.h>

static void _interrupt _far DummyInt13h (struct INTPACK);

void (_interrupt _far *GetBIOSInt13h (void)) (void)
{
    _asm {
            push    ds          // preserve ds throughout function (it is
                                 // expected to change) - pessimists may want
                                 // to preserve registers bp, di and si also

    /*   Call int 2Fh function 13h to install a dummy replacement for both
         the BIOS int 13h handler and the IO.SYS machine-specific filter.
         The function expects new addresses for these two routines in es:bx
         and ds:dx respectively, returning the previous addresses in these
         same registers. Thus, it is not possible to obtain the addresses
         without disturbing the int 13h chain. */

             push    cs
             pop     ds
             mov     dx,offset DummyInt13h
             push    cs
             pop     es
             mov     bx,dx
             mov     ah,13h
             int     2Fh

             push    es          // save es:bx on stack, to return as value of
             push    bx          // C function

    /*   Call int 2Fh function 13h a second time to reinstall the routines
         that were temporarily replaced by the dummy handler. */

             mov     ah,13h
             int     2Fh

             pop     ax          // retrieve address of BIOS int 13h handler
             pop     dx          // into dx:ax as return value of C function

             pop     ds          // restore ds = DGROUP
    }
}

static void _interrupt _far DummyInt13h (struct INTPACK r)
{
    /*   Fail the int 13h call as if for an unsupported function. */

    r.ax = 0x0100;       // return with ah = 0x01 as error code

    r.flags |= 0x0201;           // return with interrupts enabled (0x0200),
                                 // carry flag set (0x0001) to indicate error,
                                 // all other flags unchanged
    /*   For completeness, also record the simulated error code in the BIOS
         data variable (0040:0041 for floppy disks, 0040:0074 for hard disks)
         consulted by int 13h function 01h. */

    *((BYTE _far *) MK_FP (0x0040, r.dl & 0x80 ? 0x0074 : 0x0041)) = 0x01;
}

/*  There follows a small test rig. */

#include    <stdio.h>
#include    <stdlib.h>

void main (void)
{
    if (MK_WORD (_osmajor, _osminor) < MK_WORD (3, 10)) return;
    printf ("\nIO.SYS thinks that the BIOS int 13h handler is at %Fp\n",
         GetBIOSInt13h ());
}
```

The program's brief `main` function suffices for the purposes of testing and demonstration—to show, for instance, that Windows does indeed hook int 13h at the es:bx address when the FastDisk feature is configured. Typical addresses revealed by running the program in various circumstances are shown in Table 2-3.

Table 2-3: Some Typical Addresses

SITUATION	ADDRESS	COMMENTS
when Windows is not running	F000:930D	ROM BIOS int 13h handler—address recorded during IO.SYS initialization
from DOS prompt inside Windows when **SystemROMBreakPoint=off**	0781:0D60	single-byte instruction acting as breakpoint from virtual machine to int 13h handler in BLOCKDEV—byte allocated in Global V86 Data Area
from DOS prompt inside Windows when **SystemROMBreakPoint=on**	FED8:0D62	similar breakpoint from VM to BLOCKDEV—but using byte found in ROM BIOS

Since the table of sample int 13h hooks mentions the Windows SYSTEM.INI **SystemROMBreakPoint** setting, it is time for what only seems like another digression, to discuss Windows "breakpoints."

Windows Breakpoints

The GETINT13 program shows that, under Windows, int 13h is handled by a single instruction acting as a "breakpoint" from the virtual machine (VM) into the BLOCKDEV VxD, and that, with **SystemROMBreakPoint=on**, this single byte will be located in the ROM BIOS, at an address such as FED8:0D62.

Readers may wonder why Windows addresses the breakpoint in ROM with such bizarre numbers as FED8:0D62 instead of using a familiar segment like F000h or normalizing the address so that its offset component is small. Anyway, this is a good opportunity (or excuse) to talk about Windows breakpoints which, surprising as it sounds, are highly relevant to DOS internals.

A breakpoint is a specialized run-time patch that diverts the instruction stream to more substantial code. The technique is invasive, and the breakpoint must be as small as possible (a single byte) to avoid affecting anything other than the instruction whose attempted execution is to be intercepted. The usual application is for debugging, when a program's execution must be interrupted at places unanticipated by the program's design.

Windows has a similar goal—to wrest control from code designed for execution under DOS but now running in a virtual machine under Windows. Of course, Windows cannot use the int 3 dedicated breakpoint instruction (opcode CCh) available to debuggers, but instead deploys its supreme control as the ring 0 supervisor to interpret byte 63h (an illegal ARPL instruction) as a special breakpoint for Windows. When this byte is executed by a virtual-8086 task, the processor raises an invalid opcode fault which need simply be picked up by the virtual memory manager (VMM) in ring 0 and distributed to whichever virtual device asked (via the `VMM Install_V86_Break_Point` function) for the breakpoint to be installed.

This technique is so efficient at transferring control that Windows uses it not just to break into code belonging to other programs but also for new code that Windows itself places in the virtual machines. For instance, a virtual display device might want to trap a VM's int 10h chain for only a few video functions; a stub handler in the VM filters the functions of interest, but these are sent to protected mode via a breakpoint, not by complicating the DOS world with yet another interrupt-based interface to Windows.

If no stub is required, it may be preferable to reuse an existing breakpoint. More generally, if an intended breakpoint will only ever be reached as the target of a jump, call, or interrupt, then it may be better to redirect the jump, call, or interrupt to a special breakpoint set aside for just these situations. The obvious approach is for Windows to create this reusable breakpoint in much the same place it would use for new breakpoints anyway—the Global V86 Data Area, which is reserved as common memory for all virtual machines. Any convenient byte with the value 63h may be adopted, however, as long as its visibility in all virtual machines at all times is certain.

Note that 63h is the ASCII code for 'c' so that a character in a ROM copyright string might even be suitable as the location for a ROM breakpoint. If **SystemROMBreakPoint=on**, then Windows searches the ROM BIOS for a suitable breakpoint (proceeding downwards for at most 16K from linear address FFFFFh); in the case presented in the main text, a suitable byte was found at linear address FFAE2h.

A problem arises in knowing where to send control in protected mode when the reusable breakpoint is executed. There is no longer a one-to-one correspondence between a breakpoint's handler in protected mode and its linear address in the virtual machines. The answer is to associate each breakpoint handler with a different segment:offset address for the breakpoint's execution in virtual machines. In Windows language, a virtual machine asks for a V86 Call Back and is given an 8086-style far pointer to the reusable breakpoint; each request returns a different far pointer to the same byte of memory.

Returning to the example in the main text, it looks as if Windows may have started with 0857:0000 and FFAE:0002 as the breakpoint address (depending on the **SystemROMBreakPoint** setting) and that the adjustment by 00D6h paragraphs in both cases reflects the number of V86 Call Back allocations. This is roughly true, but this digression would not be complete without noting the curious way in which Windows configures its breakpoint facilities.

The Windows Resource Kit suggests that the number of breakpoints available is the value of a **MaxBPs** keyword, defaulting to 200. Obviously, Windows has to relate various pieces of information to support a breakpoint; memory for the necessary tables is set aside during VMM initialization. The memory requirement is MaxBPs * 0Ah + 0200h, but is allocated in whole pages, with the twist that the VMM configures not as many breakpoints as specified by **MaxBPs**, but as many as can be accommodated in the same number of 4K pages as would be required for exactly **MaxBPs** breakpoints (and then this value is rounded down to a multiple of 4). A practical consequence is that *all* settings of MaxBPs ≤ 358 cause the VMM to allow 356 possible breakpoints, all settings 359 ≤ MaxBPs ≤ 768 are equivalent in reality to having **MaxBPs=768**, and so forth. There is an added complication that the breakpoint tables cater for installable breakpoints (which require two entries) as well as the V86 Call Back allocations (which require only one entry each) and Protected Mode Call Backs (which are beyond the scope of this book).

The explanation behind FED8:0D62 is therefore that the VMM adjusts the normalized address FFAE:0002 for a possible 356 (or 0164h) breakpoints to get FE4A:1642 as the first address it would allocate as a V86 Call Back. Successive requests for breakpoints of any type supported in the tables cause the segment component to be incremented and the offset decreased by 10h for balance. In the case at hand, the segment had advanced to FED8h by the time BLOCKDEV sought a V86 Call Back address to pass to int 2Fh function 13h. Now, whenever an int 13h service passes through IO.SYS and then to the reusable breakpoint, the VMM knows [illegible] to consult entry number *FED8h—FE4Ah = 008Eh in the breakpoint tables* (and the FastDisk machinery can get to work).

Availability

Serious applications which make non-trivial use of the addresses returned by int 2Fh function 13h or which anticipate execution under any but the most recent DOS versions would be well-advised to test whether or not the interface is actually implemented. Certainly, the function is not available before DOS 3.10, but it seems that simply checking against the DOS version number is unsuitable; presumably, some OEM variants of IO.SYS in early DOS versions do not support the function. For instance, the WDCTRL VxD declares a requirement of DOS 3.10 or higher if it determines that the interface is not present, but actually arrives at its decision by checking that the function changes the value passed in register DS. Similarly, BLOCKDEV accepts the existence of int 2Fh function 13h only if the address passed in DS:DX is changed by the call.

Interrupt Windows

Robust users of int 2Fh function 13h have much more to concentrate on than version checking. As noted earlier, there are difficulties involved in adapting the interface to thread code into the int 13h chain rather than divert the chain to a BIOS replacement. These surface even in the sample code presented earlier. What should be a non-invasive location of points in the int 13h chain is complicated because the function requires some sort of int 13h handler in advance.

A program's first use of int 2Fh function 13h is essentially blind. At some time during the call, the int 13h routine whose address is passed in DS:DX becomes solely responsible for processing int 13h underneath IO.SYS. For a little while, this routine has nowhere else to send requests for disk access; the ds:dx and es:bx addresses returned by the function and needed for the int 13h chain's reconnection are in a sort of limbo—no longer known to IO.SYS but not yet visible to the program that requested them for its newly installed int 13h code. What happens then, if a hardware interrupt occurs and its handler uses int 13h?

Fortunately, providing for the possible use of int 13h by a hardware interrupt handler while the new arrangement of the chain below IO.SYS is unknown turns out to be more or less automatic; even for the sample code, where changes to the int 13h chain are undone immediately after int 2Fh function 13h returns, good programming practice dictates that the function be supplied with what it expects. Covering this particular interrupt window, therefore, demands no real thought from the programmer, only a provision that the int 13h handler at the address passed in DS:DX be as inert as possible until the int 13h chain has been reconnected.

Int 2Fh function 13h also expects an address in ES:BX. This, too, should belong to some sort of int 13h routine—but in this case, the address is also the one to which IO.SYS will restore the int 13h vector on receipt of int 19h. Unfortunately, a sound address for this restoration is hardly likely to be known in advance of calling the function; indeed, discovery of this address (in the hope that it is the entry point to the BIOS int 13h routines) may be the main reason for using the function. At least for the first time, there is no way to call int 2Fh function 13h with both the parameters it expects. Good programming practice is therefore frustrated—perhaps the best criterion, incidentally, for deciding whether an interface design is defective rather than just awkward.

The overloading of the es:bx pointer in IO.SYS—as both a possible point in the int 13h chain and the warm boot restoration address—means that any long-term change to the es:bx pointer (to install int 13h code as close to the BIOS as can possibly be arranged) must be accompanied by the installation of an int 19h handler that will undo the change before the IO.SYS int 19h handler attempts to restore the int 13h vector. This safeguard follows naturally when programmers think ahead about the consequences of disturbing the system, but again, the interface design complicates matters. The int 19h handler should be in place by the time that int 2Fh function 13h loads a new address into the es:bx pointer, which means that it must be installed before int 2Fh function 13h is called. However, until that function returns, the int 19h handler cannot know how to perform its supposed restoration; indeed, it will simply have to be brutal and shut down the system should some interrupting process generate an int 19h.

Preventing Interruption While Changing Public Far Pointers

Truly robust users of int 2Fh function 13h must also deal with the possibility (raised near Table 2-1) of an interruption occurring between the instructions that place a new address into either the ds:dx or es:bx pointer in the IO.SYS data segment. In these two very tiny windows, int 13h calls issued by the interrupting process will go to some meaningless address formed by combining an old segment and new offset. Given task switching, the problem of interruption at a sensitive moment is very real, not just a theoretical abstraction.

The problem is essentially the same as the familiar one of guarding against interruption while loading the ss:sp registers in two instructions. To prevent the calamity of having an interrupt handler take over with an invalid stack, the CPU is supposed to guarantee the impossibility of interruption for one further instruction after loading ss. Because some early 8088 and 8086 chips failed in this respect, it is standard practice to disable interrupts explicitly while loading ss and sp through separate instructions. It is curious that programmers writing 80386-specific code often continue the practice (unnecessarily) for ss:sp but fail to disable interrupts when writing new values into far pointers that may be used for indirect control transfer. The only difference is that catastrophe is not certain in the latter case (it occurs only if the interrupt handler tries to use the half-formed far pointer); in nature, however, the two problems are very much the same and it is unclear why they receive such different treatment from programmers.

To get some idea of how the IO.SYS routine for int 2Fh function 13h comes to exchange pointers without disabling interrupts, consider the following sample interrupt handler, which saves the address DS:DX in a far pointer for use by some other code:

```
InterruptHandler    PROC        FAR
         mov        word ptr cs:[FarPointer],dx
         mov        word ptr cs:[FarPointer + 02h],ds
         iret
InterruptHandler    ENDP
```

If this handler receives control because its address is entered in the Interrupt Vector Table, then the CPU will have cleared the Interrupt Flag already. This is a property of the way the processor transfers control in response to an INT instruction. Not surprisingly, many interrupt handlers assume this to be an entry condition for their code. It follows that when one handler "chains" to another, it should make sure that the Interrupt Flag is clear, just to reproduce normal entry conditions. Sadly, this is not always done (readers may find it reassuring to know that the `_chain_intr` function in the Microsoft C library behaves correctly).

Now consider the attitude taken by the IO.SYS code for int 2Fh function 13h. In versions before DOS 5, the int 2Fh chain ends in the DOS kernel, whose handler uses an IRET instruction to return unwanted functions. After MSDOS.SYS initialization, the SYSINIT module arranges the installation of the IO.SYS int 2Fh handler, which is written with the expectation that it must pass unwanted functions further along a pre-existing (if nearly trivial) int 2Fh chain. As with the simple example above, IO.SYS could disable interrupts explicitly for the pointer exchange, but is arguably entitled to assume that interrupts will be disabled on entry. Should this not be so, then at least IO.SYS is not directly at fault for having opened a window. The blame lies elsewhere, say with Microsoft's own ANSI.SYS or DISPLAY.SYS, both of which are certain to enable interrupts before passing int 2Fh further along the chain.

In DOS 5, the situation is slightly different. The int 2Fh routines for IO.SYS and MSDOS.SYS are installed as one handler, with a hard-coded link between the two pieces. To unify the installation of stub code in low memory for the kernel's interrupt handlers if **dos=high**, int 2Fh must be seen first by MSDOS.SYS. Some rearrangement has been necessary, with the result that in DOS 5 the IO.SYS handler for int 2Fh is the terminal one. This works in favor of IO.SYS in the sense that when **dos=high**, it may assume the A20 to have been enabled already by the DOS kernel and therefore jump straight to the HMA to process int 2Fh—without troubling over the A20 state. Another consequence however, is that IO.SYS is no longer entitled to take a view about whether the int 2Fh handler ahead of it in the chain will have disabled interrupts before passing control. Since the handler is certainly the one in MSDOS.SYS, the position should be unmistakably clear. In fact, the int 2Fh handler in the DOS kernel enables interrupts for its own work and does not disable them when passing unwanted functions further along the chain. The IO.SYS handler, however, continues to assume that interrupts will be disabled on entry. Confusion over (or ignorance of) this simple matter means that int 2Fh function 13h cannot be used in perfect safety under DOS 5 without resorting to hardware that inhibits interrupt generation.

The thinking behind this is that if reasons beyond the programmer's control mean hat the processor cannot have its attention turned away from interrupts at a sensitive ime, then the alternative is to prevent interrupts from occurring. This may be done easily nough by setting all the bits in the Interrupt Mask Register of the Programmable nterrupt Controller (PIC), which will thereafter ignore all interrupt requests. Thus, calls o int 2Fh function 13h should be surrounded by code whose effect is to disconnect the IC from external triggers. Then, when the end of the int 2Fh chain is reached and O.SYS writes new values into the ds:dx and es:bx pointers, the CPU will be accepting nterrupts, but none may possibly occur. Now consider that, in all likelihood, much more ime will be spent progressing through the various int 2Fh handlers to reach IO.SYS than an possibly be taken to execute the few instructions needed by a program to support its se of int 2Fh function 13h. Thus, if a program disables the PIC for the int 2Fh in-truction, it may as well inhibit interrupt generation until it has solved the interface's other roblems. Such an approach certainly simplifies matters for anyone intending to use nt 2Fh function 13h.

On the other hand, it is usually thought undesirable to disable interrupts for great tretches of time. Perhaps it is reasonable to say "a bug in the operating system made me o it." Even in the present case, there is a distinction to be drawn between living with a adly designed interface and overcoming some poor programming within the operating ystem. As examples of the latter come to light, they will surely be corrected (at least for uture DOS releases, unless Microsoft enjoys exposure to ridicule or intends ceding its upposed non-liability for errors in its software), but since the interface is used by rograms designed to execute under a range of past and future DOS versions, even future oftware that chooses to not support DOS 5 will probably still have to contend with this nterface's peculiar construction.

astDisk Requirements

f rather wider interest than these technical difficulties are the circumstances in which the VDCTRL VxD aborts its installation, having decided that registration with BLOCKDEV vould dislodge int 13h handlers installed beforehand (and which presumably serve some seful purpose, making it undesirable to proceed):

```
he Microsoft Windows 32-bit disk driver (WDCTRL) cannot be loaded. There is
nrecognizable disk software installed on this computer.

he address that MS-DOS uses to communicate with the hard disk has been changed.
ome software, such as disk-caching software, changes this address.

f you aren't running such software, you should run a virus-detection program to
ake sure there is no virus on your computer.
```

The address referred to in this message is none other than the one returned in es:bx by int 2Fh function 13h. For WDCTRL to not object, this address must either seem plausible as an entry point for int 13h code in ROM or belong to a handler that Windows can somehow recognize as displaceable.

Finding ROM

WDCTRL decides plausibility as an address in ROM by the simple expedient of comparing the segment component against A000h (see `Check_Hook_Addr_OK` in WDCTRL\WDRMINIT.ASM in the Windows 3.1 DDK). Remember that the PC architecture has always regarded addresses above this segment (but within the first megabyte) as available for hardware support. This is therefore the region where a ROM extension would be mapped. As far as FastDisk is concerned, an int 13h handler in a ROM extension, say for a hard drive controller, is treated on the same basis as the original int 13h handler in the ROM BIOS. The simple comparison against segment A000h means that WDCTRL makes no complaint if the es:bx address reported by int 2Fh function 13h lies in an upper memory block filled with RAM or in the HMA instead of being genuinely in ROM.

Forward Compatibility

Future users of int 2Fh function 13h who insert code into the int 13h chain via the es:bx pointer are expected to respond positively to a special interface if they mean for Windows to proceed with FastDisk installation even though this will (in effect) remove their own code from the int 13h chain. The interface follows the general scheme by which virtual devices interrogate or signal DOS software during the device initialization stages of Windows' startup. The Device Initialization Broadcast is int 2Fh function 1607h, with register BX containing an identification number for the virtual device; everything else in the protocol depends on the device. For the particular case of FastDisk, the broadcasts are issued by the devices like WDCTRL, which provide the hardware-dependent code for disk access, but since each such device would have a different ID, the broadcasts use the ID for BLOCKDEV—which is, after all, the device that actually hooks int 13h.

Table 2-4: Int 2Fh Functions for DOS Programs to Learn of FastDisk Initialization

BLOCKDEV Device Broadcast 01h—Starting FastDisk Compatibility Tests

INPUT:

AX	1607h
BX	0010h (denoting BLOCKDEV)
CX	0001h

No return values are expected, at least not by WDCTRL, which issues this broadcast and the next one merely to delimit its checks on drive compatibility. This involves reading disk sectors through int 13h for comparison with the hardware-level methods proposed for protected mode operation, so it is imperative that software be alerted to the need for int 13h to return true results. Some simple reasons for a possible disparity are that one of the sectors involved in the test is still held in memory by disk caching software; that a multi-boot program hooks int 13h to remap the disk sectors used by operating system material rather than have to move them physically; and that (dare it be said) a virus is trying to deceive int 13h callers about its presence on disk.

Incidentally, WDCTRL does not test drive controller compatibility during installation if either **WDCTRLDrive0=true** or **WDCTRLDrive1=true** in the **[386Enh]** section of SYSTEM.INI. These represent the first and second physical hard disks, respectively. Further configuration is supported through the Boolean keywords **WDCTRLAltStatus0** and **WDCTRLAltStatus1**, whose values decide whether the int 13h replacement code should read the corresponding drive controller status from port 01F7h (for false, which is the default) or, alternatively, from port 03F6h.

Table 2-4: Continued

BLOCKDEV Device Broadcast 02h—Ending FastDisk Compatibility Tests

INPUT:

AX	1607h
BX	0010h
CX	0002h

BLOCKDEV Device Broadcast 03h—Query FastDisk Installation

INPUT:

AX	1607h
BX	0010h
CX	0003h

SUCCESS:

cx	0000h

WDCTRL issues broadcast 03h only after it has begun the compatibility tests (which, remember, may be overridden) and only if the es:bx address reported by int 2Fh function 13h lies below segment A000h.

This query makes little (if any) allowance for the possibility that two or more int 13h handlers lie between the es:bx address and the BIOS entry point. By seeking a simple answer to whether or not FastDisk should be installed, the interface appears to require that one such handler answer for all. Before installing an int 2Fh handler for the query, a program that intends both to hook int 13h through the es:bx pointer and support FastDisk should first establish that the existing arrangement of the int 13h chain would have been acceptable to FastDisk (in a sense, gaining the right to speak for int 13h handlers already in the chain). If this condition is met, the program may install its int 2Fh handler, which should respond positively to subsequent FastDisk queries only after checking that there have been no further additions to the int 13h chain (and whose permission for a positive response has not been obtained).

Backward Compatibility

Int 13h handlers that pre-date FastDisk can't know of the BLOCKDEV interface; if they can be displaced from the int 13h chain with no ill effect, then Windows must recognize them explicitly. In this category, WDCTRL knows only how to determine from Quarterdeck's QEMM386.SYS (using an int 21h function 4402h IOCTL call to the QEMM386$ device) whether or not its Stealth feature is enabled. This is a "technology" by which the memory manager supplies RAM at linear addresses that are ordinarily used by ROM. It depends on trapping all execution paths into the BIOS so that the linear addresses in question can be assigned to the correct physical memory whenever (and only when) the system expects to be executing code in ROM. To this end, int 2Fh function 13h is used to divert the int 13h chain to a Stealth handler, which then acts as a gateway to the BIOS int 13h code. There is no conflict with FastDisk, since the Stealth handler will be needed only when the FastDisk devices decide that a particular int 13h request is not for them.

A Demonstration

The DISKHOOK test rig which follows uses int 2Fh function 13h to install a simple int 13h handler—as the es:bx address, where it will conflict with the FastDisk feature in Windows 3.1 enhanced mode (use the **diskhook /w** switch to enable an interface for compatibility with FastDisk).

The only visible role of this test handler is to defeat write access to the first physical hard disk (which should be noticed easily enough). DISKHOOK spawns a command processor in which to run arbitrary programs under the influence of the test code. When this test shell terminates, the int 13h handler is unhooked. This simple demonstration assumes that this removal will not be complicated by a resident program that was launched from the test shell to install another int 13h handler through the IO.SYS es:bx pointer.

```
/* DISKHOOK.C - demonstrates hooking int 13h below IO.SYS */

#include    <dos.h>
```

```
#include    <errno.h>
#include    <process.h>
#include    <signal.h>
#include    <stdarg.h>
#include    <stdio.h>
#include    <stdlib.h>
#include    <string.h>

#include    <arena.h>
#include    <intpack.h>
#include    <standard.h>

char description [] = "\n\
Demonstrates how int 2Fh function 13h hooks disk services below IO.SYS.\n\
Use with the /W switch to also enable an interface for compatibility\n\
with the Windows 3.1 FastDisk feature.\n";

char instructions [] = "\n\
Int 13h hooked below IO.SYS. To demonstrate the insertion, the handler\n\
prevents int 13h from writing to the first physical hard disk.\n\
\n\
A command processor is being started to allow investigation of this\n\
situation with whatever tools you prefer. Type EXIT when satisfied.\n";

/*  Functions and data which may be needed to force the new command
    processor to load in conventional memory  */

extern WORD _dos_getstrategy (WORD *);                 // in memstrat.c
extern WORD _dos_getumblink (BOOL *);                  // in memstrat.c
extern WORD _dos_setstrategy (WORD);                   // in memstrat.c
extern WORD _dos_setumblink (BOOL);                    // in memstrat.c

WORD wOrgMemoryStrategy;
BOOL bOrgUMBLink;

/*  Function prototypes and data declarations for interrupt handling. The
    program installs an int 13h routine below IO.SYS, but must also trap
    int 19h and may install an int 2Fh routine to resolve a conflict with
    the Windows 3.1 FastDisk feature if the /Windows switch is supplied.

    Use a typedef when it seems better to emphasise that a variable or
    function parameter is a pointer, rather than let this identity get lost
    amid all the keywords and brackets.

    The indisputably ugly prototype for the HookInt13hBelowIOSYS function
    allows all problems with interrupt windows to be dealt with inside that
    function and can greatly simplify the program's interrupt handlers. */

typedef void _interrupt _far INTERRUPT_HANDLER (struct INTPACK);

void HookInt13hAtESBXAddress (INTERRUPT_HANDLER *, INTERRUPT_HANDLER **);

INTERRUPT_HANDLER *lpOrgESBXAddress = NULL, *lpCurrentESBXAddress;
INTERRUPT_HANDLER *lpOrgInt19h = NULL;
INTERRUPT_HANDLER *lpOrgInt2Fh = NULL;

void _interrupt _far Int13h (struct INTPACK);
void _interrupt _far Int19h (struct INTPACK);
void _interrupt _far Int2Fh (struct INTPACK);

#pragma intrinsic (_disable, _enable)

/*  Function for access to the Programmable Interrupt Controller, if
    electing to deal with interrupt windows by the brutal method of using
    hardware to inhibit interrupt generation  */

#ifdef _PIC_
WORD SetInterruptMask (WORD);
#endif

/*  Flag and function for Windows support  */

BOOL bWindows = FALSE;
BOOL SupportFastDisk (INTERRUPT_HANDLER *);

/*  Exit handling, including a message to cover an obscure circumstance in
    which proper cleanup is impossible. */

int _far HardErrorTrap (void);
void RestoreInterrupts (void);
void RestoreMemoryUsage (void);

#ifndef _PIC_
char FatalInt19hMessage [] = "\n\
Int 19h warm boot attempted during window in which int 13h chain has been\n\
disconnected underneath IO.SYS and correct address for restoration is\n\
unknown. System halted - reboot computer with Ctrl-Alt-Del or reset.";
#endif

void PutError (char *, ...);
```

```
/*  ======================================================================== */

int main (int argc, char **argv)
{
    register char *ptr, *pCOMSPEC;
    int exitcode;

    /*    Int 2Fh function 13h is certainly not available before DOS 3.10. */

    if (MK_WORD (_osmajor, _osminor) < MK_WORD (3, 10)) {
          PutError ("requires DOS version 3.10 or higher");
          return (-1);
    }

    /*    Process the command line, looking for optional switches. There
          follows a boring template approach to this generally boring task,
          which receives no further comment in the book.  */

    while (++ argv, -- argc) {
          ptr = *argv;
          if (*ptr == '/') {
              ptr ++;
              if (ptr [0] == '?' AND ptr [1] == '\0') {
                    printf (description);
                    return (0);
              }
              if (strnicmp ("Windows", ptr, strlen (ptr)) == 0) {
                    bWindows = TRUE;
                    continue;
              }
              ptr --;
              PutError ("invalid switch %s", ptr);
              return (-1);
          }
          else {
              PutError ("invalid parameter %s", ptr);
              return (-1);
          }
    }

    /*    Locate the command processor. */

    pCOMSPEC = getenv ("COMSPEC");
    if (pCOMSPEC == NULL) {
        PutError ("cannot find command processor for test");
        return (-1);
    }

    /*  Make sure that the program does not terminate without restoring the
        interrupt vector chains to their original condition. That this
        program executes another would complicate any attempt to deal with
        Ctrl-C non-trivially (whatever the signal function's provisions),
        so just defeat it. */

    atexit (RestoreInterrupts);             // covers normal exit
    signal (SIGINT, SIG_IGN);               // prevents termination via int 23h
    _harderr (HardErrorTrap);               // prevents termination via int 24h

    /*  By default, the WDCTRL virtual device driver needed for the FastDisk
        feature in Windows 3.1 Enhanced mode refuses to install if it finds
        an int 13h handler already sitting between IO.SYS and the BIOS. An
        exception is offered through an int 2Fh service - to which this
        program aims to respond positively if executed with the /w switch. */

    if (bWindows) {

        /*  Get the current value of the es:bx pointer in IO.SYS by
            hooking into the chain and unhooking straight away. */

        HookInt13hAtESBXAddress (Int13h, &lpCurrentESBXAddress);
        HookInt13hAtESBXAddress (lpCurrentESBXAddress, NULL);

        /*  Reproduce the WDCTRL query (at least tolerably well). Cancel
            the /w switch if WDCTRL's installation would be prevented by
            an int 13h handler already closer to the BIOS than this
            program's handler will be. */

        if (NOT SupportFastDisk (lpCurrentESBXAddress)) {
            printf ("\nWARNING:  cannot support Windows FastDisk\n");
            bWindows = FALSE;
        }
    }

    /*  Hook int 19h before int 13h. Inserting an int 13h handler between
        IO.SYS and the BIOS has the (possibly subtle) side-effect of
        defeating the IO.SYS mechanism for restoring int 13h to its BIOS
        default in the event of an int 19h warm boot. Responsibility for
        maintaining system integrity lies entirely with the program that
        manipulates the int 13h chain at this low level. This program's
```

```
      int 19h handler makes sure that changes to the int 13h chain (if
      known to have been made) are undone before the interrupt is passed
      any futher. */

lpOrgInt19h = _dos_getvect (0x19);
_dos_setvect (0x19, Int19h);

/*    Now hook int 13h, supplying the address of the variable in which
      the original es:bx address is expected by the int 13h and int 19h
      handlers. Depending on the treatment of hardware interrupts during
      the use of int 2Fh function 13h, passing the address of the storage
      variable may allow the HookInt13hAtESBXAddress function to resolve
      internally a number of headaches caused by the way that the int 13h
      and int 19h handlers cannot act sensibly until they know the ds:dx
      and es:bx addresses. */

HookInt13hAtESBXAddress (Int13h, &lpOrgESBXAddress);

/*    Just check that int 2Fh function 13h really seems to have done
      something - apparently, some OEMs do not implement the function
      even for DOS versions >= 3.10.        */

if (lpOrgESBXAddress == Int13h) {
      PutError ("int 2Fh function 13h apparently not supported");
      return (-1);
}

/*    If co-operation with Windows is desired, hook int 2Fh.    */

if (bWindows) {
      lpOrgInt2Fh = _dos_getvect (0x2F);
      _dos_setvect (0x2F, Int2Fh);
}

/*    Before running the command processor for the test, remember to force
      the use of conventional memory. If the upper memory link is enabled
      and a loadhigh strategy set when this program starts, the new
      command processor and all its children will be loaded into upper
      memory (unless the supply is exhausted). This would limit the
      testing environment (for instance, Windows enhanced mode will not
      start if WIN386 has been loaded into upper memory, or indeed, if it
      has been loaded into conventional memory while the loadhigh strategy
      is in force). */

if (_osmajor >= 5) {
        _dos_getstrategy (&wOrgMemoryStrategy);
        _dos_getumblink (&bOrgUMBLink);
        if (wOrgMemoryStrategy != FIRST_FIT OR bOrgUMBLink) {
            atexit (RestoreMemoryUsage);
            _dos_setstrategy (FIRST_FIT);
            _dos_setumblink (OFF);
        }
    }

    /*  Now run the command processor, as described in the instructions. */

    printf (instructions);
    exitcode = spawnl (P_WAIT, pCOMSPEC, pCOMSPEC, NULL);
    if (exitcode == -1) {
        PutError ("%s", strerror (errno));
        return (-1);
    }
    else {
        printf ("\nDISKHOOK:  Int 13h hooking test completed.\n");
        return (exitcode);
    }
}

void HookInt13hAtESBXAddress (INTERRUPT_HANDLER *newhandler,
                              INTERRUPT_HANDLER **oldhandlerstorage)
{
    INTERRUPT_HANDLER *dsdxaddress;
    INTERRUPT_HANDLER *esbxaddress;

    /*  One way to deal with some design problems of int 2Fh function 13h
        is simply to prevent hardware interrupts from occurring while
        hooking the int 13h chain (note however, that this does still leave
        a problem with the non-maskable interrupt). */

    #ifdef _PIC_                               // all bits set in new mask
    WORD oldmask = SetInterruptMask (0xFFFF);  // to inhibit all hardware
    #endif                                     // interrupts

    /*  The intention is to thread the new handler into the int 13h chain as
        close as possible to the ROM BIOS. Unfortunately, key points in the
        chain are not known in advance of calling int 2Fh function 13h.
        Begin therefore, by installing the new handler as the only int 13h
        handler acting underneath IO.SYS, then rearrange the chain once more
        information is available. */
```

```
_asm {
        push    ds                          // must save ds=DGROUP

        lds     dx,newhandler               // if interrupts can occur,
        les     bx,newhandler               // then newhandler must be
        mov     ah,13h                      // able to replace BIOS for
        int     2Fh                         // a while

        mov     word ptr dsdxaddress,dx     // ds:dx and es:bx returned
        mov     word ptr dsdxaddress+2,ds   // by int 2Fh function 13h
        mov     word ptr esbxaddress,bx     // may both be needed for
        mov     word ptr esbxaddress+2,es   // chain reconnection

        pop     ds                          // restore ds=DGROUP
 }

 /*   This blind use of the interface leaves the int 13h chain is in as
      many as three disconnected pieces:

          IO.SYS directly to newhandler
          dsdxaddress  --> possibly other handlers --> newhandler
          esbxaddress  --> possibly other handlers --> ROM BIOS

      Unless the ds:dx and es:bx addresses are equal (in which case, the
      second piece effectively does not exist), the flow from IO.SYS must
      be re-routed to the ds:dx address. */

 if (dsdxaddress != esbxaddress) {
      _asm {
              push     ds
              lds      dx,dsdxaddress
              les      bx,newhandler
              mov      ah,13h
              int      2Fh
              pop      ds
      }
 }

 /*   Of course, the main reconnection is to link the new handler with the
      es:bx address. Unless hardware interrupts are inhibited, the moment
      that the new handler possesses the value currently in the local
      variable esbxaddress, int 13h requests will be able to reach the
      BIOS. Int 13h will function as before, with just the one addition
      of the new handler in front of the previous es:bx address. Note the
```

```
         importance of taking this step and the preceding one in the right
         order;   to do otherwise would risk an int 13h that skips the
         handlers between the original ds:dx address and the new handler. */

    if (oldhandlerstorage != NULL) {
         #ifndef _PIC_                          // remember to disable
         _disable ();                           // interrupts, lest an
         #endif                                 // int 13h occur while the
         *oldhandlerstorage = esbxaddress;      // pointer used for indirect
         #ifndef _PIC_                          // transfer is only
         _enable ();                            // half-saved
         #endif
    }

    /*   If however, hardware interruption is defeated, then to some extent,
         the ordering discussed above is immaterial - just as long as
         everything is in place by the time that the interrupt mask is
         restored. */

    #ifdef _PIC_
    SetInterruptMask (oldmask);
    #endif
}

#ifdef _PIC_
WORD SetInterruptMask (WORD mask)
{
    _asm {
             push    dx
             pushf
             cli
             in      al,21h     // read PIC1 Interrupt Mask Register
             mov     dl,al      //     and save in dl
             in      al,0A1h    // read PIC2 Interrupt Mask Register
             mov     dh,al      //     and save in dh
             mov     ax,mask    // write new masks to each PIC
             out     21h,al
             mov     al,ah
             out     0A1h,al
             mov     ax,dx      // retrieve original mask into ax
             popf               //     to return as value of C function
             pop     dx
```

```
#endif

void _interrupt _far Int13h (struct INTPACK r)
{
    _enable ();

    /*  Given the way that int 2Fh function 13h works, the address to which
        this trial handler transfers uninteresting requests will not be
        known until the handler has already been threaded into the interrupt
        chain.   Prevent the handler from doing anything non-trivial until
        the old address has been recorded safely. Of course, nothing need
        be done about this case if its occurrence is prevented by using the
        PIC to inhibit interrupts during the critical moments.   */

    #ifndef _PIC_
    if (lpOrgESBXAddress == NULL) {
        r.ax = 0x0100;
        r.flags |= 0x0201;
        *((BYTE _far *) MK_FP (0x0040, r.dl & 0x80 ? 0x0074 : 0x0041)) = 0x01;
        return;
    }
    #endif

    /*  For demonstration purposes, this handler simply defeats all attempts
        to write to the first physical hard disk. Everything else is passed
        on to the true BIOS int 13h handler (or to some interloper such as a
        boot sector virus that hooked int 13h before IO.SYS). */

    if (r.ah == 0x03 AND r.dl == 0x80) {
        r.ax = 0x0100;
        r.flags |= 0x0201;
        *((BYTE _far *) MK_FP (0x0040, r.dl & 0x80 ? 0x0074 : 0x0041)) = 0x01;
        return;
    }
    else _chain_intr (lpOrgESBXAddress);
}

void _interrupt _far Int19h (struct INTPACK r)
{
    _enable ();

    /*  If interrupts are allowed during the disk hook operations (i.e., not
        prevented via the PIC), then it is possible (though admittedly,
        exceptional) for int 19h to be issued while the address to which
        int 13h must be restored is not yet known. In this case, int 19h
        cannot be allowed to proceed. Make this program the current process
        for safety while writing an error message (there is a strong
        argument in favor of using the video BIOS functions for this sort
        of message, particularly to ensure that the screen is clear), then
        die. */

    #ifndef _PIC_
    if (lpOrgESBXAddress == NULL) {
        _asm {
                mov       bx,_psp
                mov       ah,50h
                int       21h
        }
        PutError (FatalInt19hMessage);
        _asm {
            die:
                jmp       die
        }
    }
    #endif

    RestoreInterrupts ();
    _chain_intr (lpOrgInt19h);
}

void _interrupt _far Int2Fh (struct INTPACK r)
{
    _enable ();

    /*  If the WDCTRL virtual device decides that es:bx returned by int 2Fh
        function 13h does not address an int 13h entry point in ROM, it
        issues a BLOCKDEV Device Broadcast to determine whether or not its
        particular Windows FastDisk service may be installed. In this
        situation, register AX contains 1607h and register BX contains the
        device ID 0010h for BLOCKDEV (the device which actually hooks
        int 13h between IO.SYS and the BIOS). The service in question is
        number 0003h, placed in register CX. If cx is not cleared to zero,
        WDCTRL aborts its installation. */

    if (r.ax == 0x1607 AND r.bx == 0x0010 AND r.cx == 0x0003) {

        /* Before returning zero in cx to let WDCTRL install, check that
           no int 13h handlers have been installed through the es:bx
           pointer since this program's int 13h handler was added. */
```

```
            HookInt13hAtESBXAddress (lpOrgESBXAddress, &lpCurrentESBXAddress);
            HookInt13hAtESBXAddress (lpCurrentESBXAddress, NULL);
            if (lpCurrentESBXAddress == Int13h) r.cx = 0;
    }
    else _chain_intr (lpOrgInt2Fh);
}

BOOL SupportFastDisk (INTERRUPT_HANDLER *current_esbx)
{
    /*  This is a reproduction of WDCTRL's own method for deciding whether
        or not to install an int 13h handler through the es:bx pointer, with
        the omission of the test for Quarterdeck's Stealth. */

    if ((_segment) current_esbx >= 0xA000) return (TRUE);

    _asm {
            mov     cx,0003h
            mov     bx,0010h
            mov     ax,1607h
            int     2Fh
            mov     ax,0001h                ; TRUE if cx returns as zero
            jcxz    done
            dec     ax                      ; FALSE otherwise
        done:
    }
}

void RestoreMemoryUsage (void)
{
    _dos_setstrategy (wOrgMemoryStrategy);
    _dos_setumblink (bOrgUMBLink);
}

void RestoreInterrupts (void)
{
    if (lpOrgESBXAddress != NULL)
        HookInt13hAtESBXAddress (lpOrgESBXAddress, NULL);
    if (lpOrgInt19h != NULL) _dos_setvect (0x19, lpOrgInt19h);
    if (lpOrgInt2Fh != NULL) _dos_setvect (0x2F, lpOrgInt2Fh);
}

int _far HardErrorTrap (void)
{
    return (_HARDERR_FAIL);
}
```

```
void PutError (char *str, ...)
{
    va_list va_start (argptr, str);
    printf ("\nDISKHOOK error:  ");
    vprintf (str, argptr);
    printf ("\n");
    va_end (argptr);
}
```

Because of the desire to demonstrate WDCTRL's interpretation of upper memory as ROM, the DISKHOOK program has to manipulate the DOS memory allocation strategies and upper memory linkage—so that although DISKHOOK may have been started with a loadhigh strategy, its test shell will have the strategy and UMB linkage restored to their defaults. Without this provision, it would not be possible to run Windows enhanced mode from within the test rig. The little extra code involved is separated into the following file—a collection of simple C functions for working with memory allocation strategies and UMB linkage, which will be used in several other programs later in this book.

As a rule, whenever this book dresses a DOS function in C language for public access, it will follow the style employed by the Microsoft C library for its `_dos` family of functions, whose prototypes are given in the DOS.H header file. These return the DOS error code on failure and zero on success. Meaningful quantities returned by the DOS function are returned to the C caller by storing them at addresses passed as function arguments.

```
/* MEMSTRAT.C - C functions to control DOS memory allocation */

#include    <standard.h>

WORD _dos_getstrategy (WORD *stratptr)
{
    WORD strategy;
    _asm {
            mov     ax,5800h
            int     21h
            jc        done
            mov     strategy,ax
    }
    *stratptr = strategy;               // for memory model independence
    _asm {
            xor     ax,ax               // return zero for success
        done:                           // else error code already in ax
    }
}
```

```
WORD _dos_setstrategy (WORD strategy)
{
    _asm {
             mov     bx,strategy
             mov     ax,5801h
             int     21h
             jc        done
             xor     ax,ax
         done:
    }
}

WORD _dos_getumblink (BOOL *umblinkptr)
{
    BOOL umblink;
    _asm {
             mov     ax,5802h
             int     21h
             jc        done
             xor       ah,ah
             mov     umblink,ax
    }
    *umblinkptr = umblink;
    _asm {
             xor     ax,ax
         done:
    }
}

WORD _dos_setumblink (BOOL umblink)
{
    _asm {
             mov     bx,umblink
             mov     ax,5803h
             int     21h
             jc        done
             xor     ax,ax
         done:
    }
}
```

Creating the test rig is a simple matter of compiling both source files and linking them together. The main module has been written with the expectation that the finished program will be called DISKHOOK.EXE.

Readers with a write-behind disk cache can easily establish that int 2Fh function 13h installs the program's test handler at a lower level in the int 13h chain than would be achieved by simply hooking the interrupt vector. In the latter case, the test code would intercept an attempted write before any other int 13h handler—so the request would simply be refused. Under the environment created by the test rig, however, an attempt to write to the affected disk is dealt with first by the write-behind disk cache, which allows the caller to go on with other work assuming the disk operation was an immediate success. In reality, the int 13h request has merely been set aside for completion at a more convenient time. When this moment arrives, the cache passes the write request to lower levels, only to find that the test code (installed at very nearly the lowest possible level) fails it. Users of SMARTDRV.EXE are cautioned to try the experiment only when the computer's shutdown will not be troublesome. This particular cache offers to retry delayed write operations that encounter errors but gives no other option, effectively hanging the machine. This also demonstrates one of the less well-publicized hazards of write-behind disk caching: Applications may think their file operations have been successful and be unable to recover properly (for instance, to repair a half-written database file) when a low-level disk error comes to light only later.

For evidence that the test handler really has become the new es:bx address, readers may follow the int 13h chain with a debugger, use the `GetBIOSInt13h` test function from earlier or take an indirect inference from the behavior of Windows enhanced mode when launched from within the test rig. Assuming that WDCTRL is configured to check drive compatibility during installation, the fact that DISKHOOK has already installed an int 13h handler at the es:bx address means that WDCTRL will complain during Windows' startup unless special steps are taken (see Table 2-5).

Table 2-5: WDCTRL and DISKHOOK Interaction

INSTALLATION COMMAND	WDCTRL INSTALLS?	EXPLANATION (REACTION TO TEST HANDLER)
diskhook	no	treated as unknown disk software
diskhook /w	yes	recognized as displaceable (assuming **/w** switch's support for FastDisk query is not cancelled by detection of uncooperative handler already closer to BIOS)
loadhigh diskhook [/w]	yes	treated as if in ROM (provided enough upper memory is free for **loadhigh** to arrange segment address ≥ A000h)

In the cases where WDCTRL installation succeeds, requests for access to the first physical hard disk (if this has FastDisk support) will be diverted to BLOCKDEV's int 13h handler in protected mode and will never reach DISKHOOK's test handler. A practical consequence is that write access is possible from a DOS Prompt opened within Windows, where it was not possible before Windows was started.

The machinations to share responsibility for a positive response to the FastDisk query may be tested by nesting DISKHOOK's execution before attempting to start Windows. This is shown in Table 2-6.

Table 2-6: Nesting DISKHOOK's Execution

FIRST	SECOND	WDCTRL INSTALLS?	EXPLANATION
diskhook	diskhook /w	no	During installation, second instance notices that first does not respond to FastDisk query; therefore, second installs without supporting FastDisk.
diskhook /w	diskhook	no	On receipt of FastDisk query, first instance detects presence of second (which has evidently not installed its own FastDisk support) and therefore responds negatively.
diskhook /w	diskhook /w	yes	During installation, second instance notices that first supports FastDisk and therefore takes over responsibility for responding positively in future.

Straightforward adoption of the BLOCKDEV interface would have allowed WDCTRL to install in each of these three cases. This would be harmless here, but consider that without its **/w** switch, DISKHOOK stands in for any pre-FastDisk client of int 2Fh function 13h; if Windows thought it safe or reasonable to displace these without their acquiescence, it would not waste time and effort raising the query. With the **/w** switch, DISKHOOK serves as a model FastDisk-aware user of int 2Fh function 13h, in that it refrains from declaring support for FastDisk if it knows of possibly incompatible handlers between the es:bx address and the BIOS. A positive response to the FastDisk query despite the presence of such other handlers would of course be justified by explicit user-level direction—either as a configurable option in a program that intends to hook into the int 13h chain anyway or as a self-standing program to be loaded when the user knows that supposedly "unrecognized disk software" actually presents no conflict.

Although Windows enhanced mode may be run from within the test rig, it is imperative that the reverse not be attempted. Indeed, the test rig must not be run under the supervision of *any* task switcher, for although these localize the Interrupt Vector Table, changes made further along the int 13h chain will almost certainly be treated as global. In other words, the switcher restores the int 13h vector to the value it had when the incoming task was last executing, but further along the chain the es:bx pointer in IO.SYS retains the address loaded by DISKHOOK—which will have lost all relevance when DISKHOOK was swapped (or mapped) out of memory for the task switch.

An important question is whether the DISKHOOK program should include code to prevent its execution under either a task switcher or Windows enhanced mode. Curiously, the program could be adapted to live with the cooperative type of task switcher (detect task switch and unhook int 13h code). Supporting Windows enhanced mode is *not* possible unless a VxD has already been loaded—the VxD scheme assumes that the only possible task switching clients are global; this therefore places the responsibility on Windows to instance *all* sensitive system material. Note the similarity to Chapter 14's discussion of extending XMS: Task switchers must instance the end of the XMS chain to support local additions (Windows 3.1 does, but 3.0 doesn't).

Note finally that if a given system can be rebooted successfully via int 19h before DISKHOOK's use of int 2Fh function 13h, then it can also be rebooted afterward. This is not true of the way that int 2Fh function 13h is used when Quarterdeck's Stealth facility hooks the int 13h chain ahead of the BIOS entry point. Nor is it true of the way in which BLOCKDEV diverts the int 13h chain to a break point for the transition to protected mode—but it hardly matters, since Windows destroys int 19h in several other ways.

The Politics of Safety and Compatibility

This apparently obscure interface from DOS's lowest reaches turns out to be a rich microcosm of the PC programming industry. Many of the most significant political questions have a place here. Take the matter of compatibility.

Just at the DOS level, it has been seen that Microsoft has taken deliberate steps in DOS 5 to accommodate some program that relies on the transfer address (the ds:dx address) used by the IO.SYS int 13h handler to be stored at 0070:00B4. This happens to be true for DOS 3.30 and 4.01 but is clearly an internal matter highly liable to change not just with future DOS versions but also from one OEM variant of IO.SYS to another. Worse, the assumption was implemented despite the existence of an interface—undocumented, admittedly, but an interface nonetheless and inherently more stable than direct access to data. Perhaps there were compelling reasons why the program's designers preferred direct access to the IO.SYS data rather than use the interface. On the other hand, it may be that they simply didn't know of the interface. Just in case the author's prejudices on this subject aren't already clear: It seems all too common for undocumented features to be described and built into commercial programs before they have been properly understood.

Is it a proper public concern to identify software vendors who have engaged in some utterly insane practice despite their claims of applying the highest possible standards?

That int 2Fh function 13h turns out to be relied on by Windows 3.1 is an excellent example of a seemingly marginal interest or practical value becoming important at a later date. When a new version of Windows (or more likely, some future development such as NT) can support a given DOS feature in only a restricted sense, there is the first question of whose preexisting software receives explicit support (by way of separate treatment, application of run-time patches, etc.) in the new Windows version, and a second question of how best to guide future DOS applications to use the feature in a compatible manner.

For an instance of the first question—whose software receives explicit support from Microsoft?—the FastDisk example shows explicit support being given to Quarterdeck's Stealth. Quite possibly, Microsoft would have accommodated other cases but was simply unaware of any other software with which FastDisk might conflict. Is it a proper matter of public interest to know the machinations behind such support decisions?

For the second question—what advice to give DOS programmers?—consider the case of a low-level DOS programmer who has a moderately successful product, perhaps even one that works correctly at close inspection. At some stage, his customers start complaining that Windows will not run while his product is installed. Doubtlessly, Windows will supply an interface through which the conflict may be resolved, but at least in the case of FastDisk, the DOS programmer has to look to the Windows Device Driver Kit (at some not inconsiderable cost) for scant documentation about how to fix his problem. It has been seen with int 19h that some low-level matters become foreign to Windows developers, so is it reasonable to expect the DOS programmer to understand what Windows is doing? Add to this that the programmer in question will be none too happy at having his otherwise sound product suddenly seem unattractive to his customers when perhaps some bigger fish are accommodated despite misuse of the facility that is now incompatible with Windows. Is this just life at work? Is it a matter for public scrutiny?

Moreover, in the case of FastDisk, it was seen that the interface actually supplied for resolving conflicts is minimal, offering no arbitration besides that which FastDisk-aware respondents are able to deduce on their own. It was seen that there exists a solution whose implementation by one program cannot make matters worse and whose implementation by all would solve the problem perfectly. The situation is like a party game sometimes used to introduce courses on Game Theory. Imagine that all the people in a room are asked to pick a number between 0 and 100 (inclusive) and that a prize will be given to the one whose number is closest to two-thirds of the average. Now, if everyone present thinks the problem through and believes that everyone else will do likewise, they will all choose 0 and share the prize. In practice, however, the result depends on the collective estimation of everyone's interest in analyzing the problem and their ability to do so. There is no place for this guesswork in professional software development.

These shortcomings in Microsoft's use of int 2Fh function 13h for Windows FastDisk may matter still more in the political context, not least because Microsoft documentation (the Windows Resource Kit) promotes the WDCTRL source code (available in the Windows 3.1 DDK) as a template for use by vendors of third-party FastDisk devices and expresses concern about others' standards of safety. Unless Microsoft has a sense of tongue-in-cheek humor rarely seen in large corporations, it ought to direct this concern inward.

Windows Support

The IO.SYS int 2Fh handler acquires two new functions in DOS 5. One of these, function 16h, is the IO.SYS contribution to the DOS system files' support for Windows. Deep examination of how the DOS 5 system files accommodate Windows is deferred to the next section's analysis of the DOS kernel, not just because IO.SYS plays the smaller role but because it is the DOS kernel which speaks for both system files in situations where Windows expects a joint response. On two counts, however, IO.SYS acts for itself; the discussion in the next section will be less cluttered if these are dispatched here.

Error Conditions in Critical Sections

IO.SYS monitors the Windows startup and exit broadcasts (int 2Fh functions 1605h and 1606h, respectively) so that it knows when Windows enhanced mode is running. The intention is to open a special avenue of communication to help the DOSMGR device with the two cases in which IO.SYS seeks user-level interaction.

These events occur when the **stacks** system overloads and when the default block device driver is asked to access a unit that is marked as inactive in the logical drive map. Ordinarily IO.SYS responds by displaying the messages

```
Internal stack overflow
System halted
```

and

```
Insert diskette for drive A: and press any key when ready
```

With enhanced mode's preemptive multi-tasking of virtual machines (its ability to schedule DOS applications in the background without requiring their assistance), it becomes possible for a VM to trigger these situations when it does not have the focus, for instance when it is not perceived as the foreground task for purposes such as keyboard input or access to the display. Much the same may happen when a DOS session is run in a window: The System VM has the focus.

The problem for Windows is that these messages emanate from so deep within the system's processing that Windows is unable to switch between virtual machines (at least, not to use DOS services in another VM). The focus must be restored to the VM in the critical section so that the matter can be resolved, or, in the case of the fatal stack overflow, so that the user can at least know what has happened.

The DOS 5 version of IO.SYS knows to call int 2Fh function 1684h for VxD ID 15h to find the calling point for the DOSMGR device's V86 API, then to use DOSMGR function 01h to set the focus. As explained earlier, the calling point is a Windows breakpoint. It is not clear why IO.SYS returns to low memory in order to call this breakpoint. Presumably the caution has some reason—as if Windows (perhaps in an old version) cannot be trusted to resume execution at an address in the HMA. It appears that the concern is not that the DOSMGR call might not preserve the A20 state: Alone among the cases in IO.SYS where the instruction stream moves from low memory to the HMA, no attempt is made to check the A20 status (to enable it, if necessary) on the way.

Instance Data

The interception of the Windows startup broadcast also gives IO.SYS the opportunity to inform Windows about data that requires instancing. To understand the requirement, consider how DOS reads special keys from the CON device.

Special Keys

An occasionally frustrating feature of console input under DOS is the representation of certain keys (function keys, cursor keys and various combinations involving Ctrl or Alt) as a sequence of two characters: first a null byte and then a second character to identify the keystroke.

If circumstances oblige a program to wait for a key, the extended representation is just an inconvenience and may be overcome by defining a function to read the console twice if necessary and combine the results into an integer:

```
#include    <conio.h>

int GetKey (void)
{
    int ch;
    return ((ch = getch ()) ? ch : - getch ());
}
```

Programs that don't allow for this produce strange effects. Suppose for instance, that the user has been prompted to "press any key." If, by perversion or by accident, the user chooses Ctrl-F4, then a single call to `getch` will return a null byte. Further down the system, the CON driver will have set aside the keycode 0x61 to be returned when DOS next asks for input (note here that the `getch` C library function is essentially a front end for DOS function 08h). If the program's routine to wait for any key makes just one call to `getch` and makes no special provision to drain the second byte if the first is null, then a little later, the user may not be happy that the program insists he has entered the letter 'a'.

But even if a program does allow for two console reads, there is no guarantee that they will execute uninterrupted. Suppose that a user has entered Ctrl-F4 intentionally, but that a supervising program like Windows swaps tasks after the first `getch`. The CON driver has a keycode pending, which Windows will have to clear away to avoid confusing the new task. On return, the user will find that Ctrl-F4 has gone missing and that the program is still waiting for input, which will in any case now be interpreted incorrectly.

Things are better in DOS 5, since system components (such as the various CON device drivers and the DOSKEY command line editor) that buffer input for other programs are able (by hooking the int 2Fh function 1605h initialization broadcast from Windows, and/or the int 2Fh function 4B05h task switcher Identify Instance Data broadcast) to advertise the existence of these buffers as "instance" data that should be saved and restored across a task switch.

Special Keys, Revisited

Clearly a programming headache is revealed here, without yet touching the question of non-destructive reads. These are what DOS itself has to do when processing special keys like Ctrl-C, Ctrl-P, and Ctrl-S. In these situations, DOS has to peek at standard input to see if one of the special characters is waiting. The catch is that it can look ahead only one byte at a time, leaving the writers of DOS with the same problem as other programmers: they must take great care to watch for null bytes in order to distinguish normal ASCII characters from the extended keycodes.

Users of DOS 3.30 may have experienced problems using Ctrl-P, which is meant to toggle printer echo on or off but sometimes has no effect. The following short program may help a closer investigation of this problem.

```
#include     <conio.h>
#include     <stdlib.h>

main ()
{
    register int ch;
    register char *ptr;
    char buf [8];

    for (;;) {

     /*  Read a byte from the console and convert it to a string of
         hexadecimal digits for display (between square brackets). */

     itoa (ch = getch (), ptr = buf, 0x10);
```

```
        putch ('[');

        do {
            putch (*ptr ++);
        } while (*ptr);

        putch (']');

        /*  Begin a new line, anticipating the next character. Note though
            that if getch () has returned a null byte, then the next byte
            returned by getch() will be the second half of a keycode, in
            which case a new line is not wanted. */

        if (ch) {
            putch ('\r');
            putch ('\n');
        }
}
```

The program displays the hexadecimal values corresponding to successive keystrokes, starting a new line for each new key. Numbers are enclosed in square brackets, so that Alt-R, for instance, should produce [0][13], since 0x13 identifies the key marked 'R'. This does indeed happen if the program is run under DOS 4 or 5.

But under version 3.30, only the [0] is displayed, after which the program waits for more input. The reason for this is that although DOS 3.30 watches for null bytes, it does not maintain its records properly, leading it to mistake the 0x13 from Alt-R for the ASCII character corresponding to Ctrl-S. Curiously, the flag set by DOS 3.30 on receipt of a null byte is used only to distinguish the second byte (0x10) of Alt-Q from the ASCII value for Ctrl-P and is only cleared when DOS thinks it has read Alt-Q. Thus, in version 3.30, once print echo has been turned on by Ctrl-P, a second Ctrl-P will turn it off only if there have been no intervening extended keys or if the last of them happens to have been Alt-Q.

Cynical readers will have gathered that the problems do not stop with version 3.30, indeed, not even with version 5. Let us return to Ctrl-S, to which DOS responds by pausing execution until more input is received. In all recent DOS versions, this expectation will be satisfied by a null byte, so that if the user follows Ctrl-S with Ctrl-F4 (as in the wait for any key scenario), a letter 'a' will be left waiting in the CON driver. This equivalence, along with many others, may be verified even from the command line.

Readers may wonder why the loop in the keycodes fragment has no explicit exit. Of course, the program can be terminated with Ctrl-C or Ctrl-Break: This is

> undeniably ugly, but the instructive point (and an exercise for the reader) is that the program can also be stopped with Ctrl-2.
>
> The problem of requiring two DOS calls to retrieve a "key" has several edges. I'm not sure the puzzle is two-dimensional, let alone suitable for squeezing into a book: There is the problem of mishandling by the DOS kernel (even in DOS 6, Ctrl-S, Ctrl-F4 is indistinguishable from 'a').
>
> Of course, once you get on to instancing devices, you have the connection with the call made by the Windows DOSMGR VxD (int 2Fh function 1607h bx=15h cx=5) to find the device size. There are all sorts of minor glitches that can occur because of differences in the various OEM implementations of the LoadHi virtual device.
>
> Well, it gets worse.

There are the implications for DOS Shell and Windows standard mode task switching since DOSSWAP/DSWAP will lose the pending byte if a task switch is requested between the two DOS calls.

In Windows enhanced mode, the byte in IO.SYS that holds the pending scan code is instanced. Indeed, reorganization of IO.SYS for DOS 5 leaves this holding byte for the default CON device driver as the only instance data in the file. But what if you add another CON device that isn't Windows-aware?

Windows gets around this by setting Local=CON in SYSTEM.INI, which means that the whole CON device driver is instanced. Problem: if you load ANSI.SYS and DISPLAY.SYS, which of them gets instanced? Answer: only the one you load last. And DISPLAY.SYS has nothing to do with the two-call retrieval of extended keystrokes.

This is all related to interrupting ANSI escape sequences, incidentally. If you load ANSI before DISPLAY, then in Windows enhanced mode, set PROMPTs with different colors in two DOS boxes, you will see that using the prompt in one box changes the color for the next command issued in the other DOS box!

Alternatively, if you load DISPLAY first, then you can end up with silly situations where (in full screen) MODE tells you the codepage is 437, but typing Alt+155 gives you a Danish o-slash rather than a cent symbol.

Subletting the HMA

The remaining int 2Fh function defined by IO.SYS in DOS 5 is the interface through which the HMA (obtained as a single unit, typically from an XMS server) is distributed among various DOS components when **dos=high**. The IO.SYS segment intended for relocation to the HMA is designed for loading at the 1MB boundary, where it will be followed by the DOS kernel's code segment and perhaps also by a piece of HIMEM.SYS, code pages prepared for DISPLAY.SYS, the collection of **buffers** which streamline the DOS kernel's disk access and a shared segment of COMMAND.COM code (discussed later in this chapter). It is only natural that the first component placed in the HMA should also supply the interface through which others may join it.

Table 2-7: IO.SYS Interface for Apportioning DOS's HMA Allocation

Int 2Fh function 4A01h—Query HMA Space

INPUT:

AX	4A01h

SUCCESS:

bx	bytes free
es:di	address of free space—based with respect to FFFFh

FAILURE:

bx	0000h
es:di	FFFF:FFFF

Int 2Fh function 4A02h—Allocate HMA Space

INPUT:

AX	4A02h
BX	bytes required

SUCCESS:

bx	bytes allocated—equal to the bytes required, but rounded up to whole paragraphs
es:di	address of allocated space—based with respect to FFFFh

FAILURE:

es:di	FFFF:FFFF

In the present implementation, space in the HMA is allocated in whole paragraphs. A private variable in the IO.SYS data segment holds the address (with respect to segment FFFFh) of the first free paragraph. This marker is advanced with each allocation. There is no provision for DOS components to release their HMA space.

There are two distinct behavioral phases to these functions separated by the moment when SYSINIT becomes formally ready to run programs through the CONFIG.SYS **install** statement. Prior to this event if the HMA modules of IO.SYS and MSDOS.SYS are not already in place in the HMA when either function is called, then the necessary relocation is attempted before proceeding. The whole function fails if the relocation cannot be completed, most obviously if SYSINIT is not intending to configure **dos=high** but possibly also because no XMS interface is yet available or because the HMA either does not exist or is somehow rendered unusable (say, by the presence of a VDISK header). Once it becomes formally possible to execute programs, the functions simply fail if the system files' segments are not already in the HMA.

The last point may require a little explanation in advance of Chapter 3 (which traces the system's initialization in some detail) and Chapter 10 (which examines the way an XMS server claims a pool of extended memory, to manage on behalf of XMS clients). SYSINIT is unable to configure **dos=high** until an XMS server has been activated. The problem is that SYSINIT cannot know the identity of the XMS server in advance. After each device driver initialization and, if necessary, after processing all the **install** statements, SYSINIT checks for the presence of an XMS interface, so that it may move the system files' code segments to the HMA as soon as possible after the HMA becomes available. How then is HIMEM.SYS (the extant XMS server) able to use the HMA for some of its own material? If the system code segments must be in the HMA first, but SYSINIT cannot arrange this until it has already loaded an XMS server, how does the XMS server also use the HMA?

Measuring DOS's HMA Usage

Although it is not this book's habit to present programs which merely rephrase the tabular presentation of an interface or structure, the following short piece of code corresponds to Table 2-7 and ascertains the extent of DOS's HMA usage and serves either as counterpoint to or confirmation of the occasional suggestion by reviewers that DOS 5 obtains no particular advantage from the HMA.

```
/* DOSINHMA.C */

#include    <stdio.h>
#include    <stdlib.h>

#include    <standard.h>

struct DOSINHMA {
    WORD wFreeBytes;
    void _far *lpFreeSpace;
};

void GetDOSHMAState (struct DOSINHMA *);

void main (void)
{
    struct DOSINHMA dosinhma;
    register WORD bytes_free, bytes_used;
    if (_osmajor < 5) {
        printf ("\nRequires DOS version 5 or higher\n");
        return;
    }
    GetDOSHMAState (&dosinhma);
    if (dosinhma.lpFreeSpace == (void _far *) -1) {
        printf ("\nDOS is not using the HMA\n");
        return;
    }
    else {
        printf ("\nDOS is using the HMA");

        bytes_free = dosinhma.wFreeBytes;
        printf ("\nFree space is %d bytes", bytes_free);

        bytes_used = -bytes_free;  // 0FFFFh bytes total
        printf ("\nUsage is %u bytes (%d%%)\n", bytes_used,
           (WORD) (((DWORD) bytes_used * 100) >> 16));
    }
}

void GetDOSHMAState (struct DOSINHMA *resultptr)
{
    struct DOSINHMA dosinhma;
    _asm {
           mov      ax,4A01h
           int      2Fh
           mov      dosinhma.wFreeBytes,bx
           mov      word ptr dosinhma.lpFreeSpace,di
           mov      word ptr dosinhma.lpFreeSpace+2,es
    }
    *resultptr = dosinhma;
}
```

For instance, with a CONFIG.SYS file containing the lines

```
buffers=10
device=c:\dos\himem.sys
device=c:\dos\display.sys con=(ega,437,(1,3))
dos=high
shell=c:\dos\command.com c:\dos /p
```

the dedicated portions of IO.SYS, MSDOS.SYS, COMMAND.COM and HIMEM.SYS, plus 10 **buffers** and code page support for a VGA (bitmaps for all 256 characters in resolutions of 8, 14 and 16 raster lines per character cell) amount to seven-eighths of the HMA:

```
DOS is using the HMA
Free space is 8464 bytes
Usage is 57072 bytes (87%)
```

Moreover, filling the HMA is well within the range of DOS's configurability. Setting **buffers** to 20 is an arguably optimal value in the absence of disk caching software. Users in the habit of loading hundreds of files into a directory may easily need more.

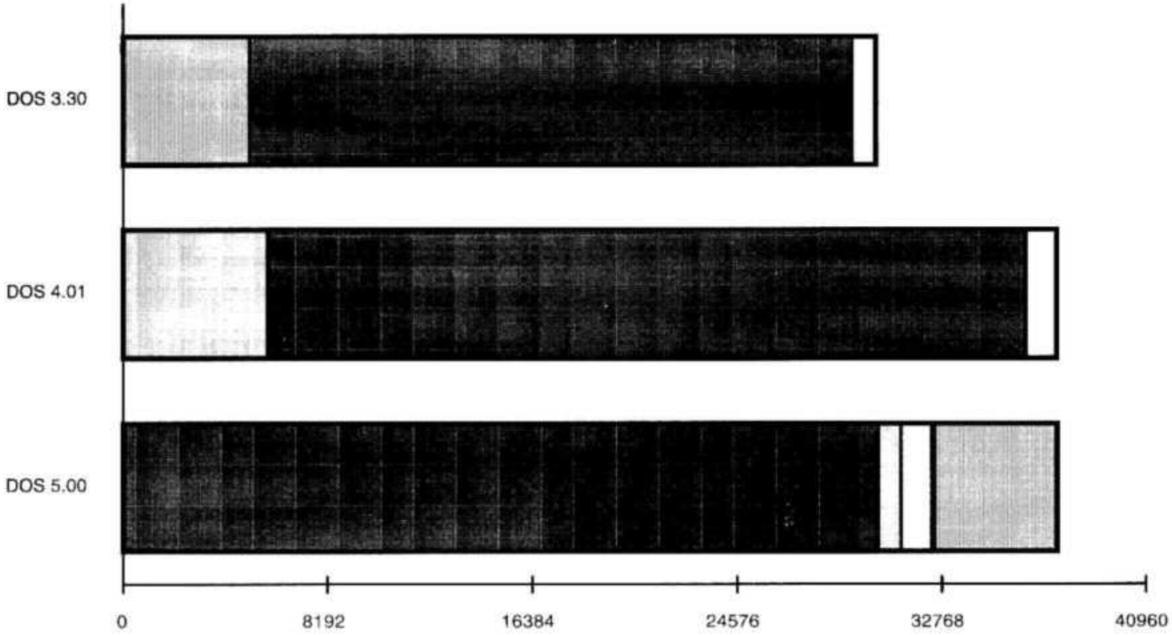
MSDOS.SYS—The DOS Kernel

The MSDOS.SYS file supplies the whole of DOS's interface with applications—a hundred or so int 21h functions that present programs with a set of standard tools for using the computer. The kernel manages memory and defines processes. Perhaps most importantly, it organizes a file system to support device-independent access to hardware. The narrow channel of the device-driver layer—with its picture of hardware as either a stream of characters or a collection of sectors—has to be gathered into a logical structure closer to the needs of applications. In addition to the built-in system, seen most conspicuously in the FAT system required for block storage media, the kernel also provides a way to add both alien file systems (such as might be created for the special needs of write-once media) and systems that have their own device driver channel (say, on another machine). Above this sits an integrated directory system that lets applications find and use files no matter how they are organized on their storage media or pulled together from a character stream.

All this requires a sizable implementation. Not surprisingly, the kernel is by far the larger of the two system files in terms of its memory footprint. Figure 2-3 depicts the kernel's layout as a file on disk for recent DOS versions, again by shading the areas which represent material retained permanently. Regions of medium shading are for kernel data and the darkest shading denotes kernel code (easily the bulk of the file).

In versions before 5.0, the MSDOS.SYS file had no special large-scale organization and no segmentation. A data area at the start of the file is followed by code, of which a small portion at the end is needed only for the kernel's initialization (indicated by a white space in the diagram). In DOS 5, all the resident code (approximately 30K) is relocated to the High Memory Area if a **dos=high** regime is adopted. The light shading appearing only in the DOS 5 element of Figure 2-3 represents a small amount of code that is also transferred to the HMA when **dos=high**, but which is not needed otherwise. It deals with problems peculiar to having the A20 almost pemanently enabled when DOS operates in the **dos=high** configuration.

Figure 2-3: Structure of MSDOS.SYS on Disk—Comparison of Different Versions



Low and High Segments Revisited

As with IO.SYS, the segment that must lie in low memory also contains a little code. There are the expected stubs for the kernel's non-trivial interrupt handlers (00h, 20h, 21h, 25h, 26h, 27h, and 2Fh) and also for CP/M-style DOS calls.

Most other interrupts from 20h to 3Fh inclusive, are pointed to an IRET instruction in low memory. The exceptions are that int 29h is set up by IO.SYS and that the space in the Interrupt Vector Table for interrupts 30h and 31h is used for the CP/M-compatible DOS interface.

In addition to stub code, the MSDOS.SYS data segment holds the NUL device driver's strategy and interrupt routines (necessarily in the same segment as the device driver header) and a default routine for mapping extended ASCII characters to upper case.

For all these, DOS must check that the A20 is enabled before transferring to more substantial code in the HMA. As a general rule, when these interrupts return from the HMA, the A20 remains enabled.

Again following IO.SYS, each low-memory stub compares the eight bytes at addresses 0000:0080 and FFFF:0090 to determine whether or not the A20 is already enabled. If so, then control may be transferred to the HMA forthwith. If not, then DOS asks the XMS server to enable the A20.

DOS also has stubs for returning to low memory before issuing interrupts 23h, 24h, and 28h. There is also the case of going back to low memory (and disabling the A20, no less) before transferring control to a child program; see Chapter 5.

Note too that DOS uses the critical stack while enabling the A20. This applies to all DOS services at kernel level (e.g., interrrupts 25h, 26h and 2Fh)—even the normally reentrant int 21h functions such as 50h (Set PSP) and 51h (Get PSP). Everything would be fine except that the stub assumes interrupts are disabled on entry. If they are not (because of someone else's bug) then what sort of catastrophe is the system exposed to? Examples of programs that may chain int 21h without disabling interrupts include the new SMARTDRV.EXE disk cache. Examples for int 2Fh were discussed earlier in this chapter.

ROM-Executable DOS

It has been noted several times already that the large amount of publicly accessible data in the system files means that the separation of system material into private and public to support execution in the HMA is practically the same as the division into read/write and read-only required for a ROM-executable DOS.

In the ROM-executable DOS version, the IO.SYS data segment reserves the word at offset 0Ah for the segment address of the first ROM module containing DOS programs.

The DOS kernel may hold a far pointer to an ROM Exclusion List. Currently, this is at offset 12BAh in the ROMDOS.SYS data segment. The far pointer is null if no list is loaded. The list starts with a word which gives the number of four-byte entries to follow. Each entry consists of two segment addresses describing a range (inclusive) of memory exempted from ROM scans.

Windows expects the first word of the IO.SYS data segment, that is, the word at 0070:0000, to contain the segment address of the ROM, which must in turn pass the standard validity test for an external ROM. Thus, the first word in the ROM must be the signature AA55h, with the next byte giving the size of the ROM, measured in units of 512 bytes. A simple checksum of all bytes in the ROM must yield zero.

ROM Identification

DOS expects a ROM module to start with the structure

OFFSET	SIZE	DESCRIPTION
00h	word	signature AA55h
02h	byte	size of ROM measured in multiples of 512 bytes; DOS checks for ROMs only at 2K boundaries

At offset 03h, there should be code with which the BIOS may initialize the ROM module before loading the operating system from disk.

Starting at offset 06h, DOS expects a list giving the names and entry points of programs in the ROM. The list is terminated by an entry the length count of whose name is zero. It is also considered to finish if it either overruns 64K or runs into a region excluded from consideration (but the coding does not seem to be consistent).

OFFSET	SIZE	DESCRIPTION
00h	byte	length of name
01h		name
	03h	calling point—three bytes allowed for JMP instruction

This is the same scheme understood by COMMAND.COM (at least from DOS 3.30) for executing ROM modules on PCs with model byte FDh.

Programs in ROM

The ROM version of DOS provides two functions (int 21h functions 6Dh and 6Eh) for locating ROM programs—using a scheme strikingly similar to DOS functions 4Eh and 4Fh. The structure returned in the DTA if the first search succeeds is a little different.

OFFSET	SIZE	DESCRIPTION
00h	0Dh	12-byte name of ROM program, followed by null terminator
0Dh	dword	address at which to resume search; three bytes after entry point to desired ROM program
11h		ASCIIZ mask used in search

The mask and the address must be retained intact if the subsequent searches are to proceed in the expected, orderly manner. DOS has no length restriction on the mask. This is not a real problem, since the mask is the string with which the caller seeds the search. More troublesome is that DOS performs no check on the length of a ROM program's name. The corresponding field in the DTA allows only enough space for an eight-byte filename and three-byte extension with a period to separate them, so it is conceivable that an errant ROM module or fluke match could cause meaningless information to be loaded into the DTA.

The mask may contain wildcards. A question mark matches any single character other than a period. An asterisk in the mask matches all the remaining characters in a name. Characters after the asterisk are ignored, with the exception that an asterisk must be matched with an asterisk, which resynchronizes the comparison. The coding does not quite reproduce the usual behavior; for instance, the mask "*a." matches any string (and only those strings) ending in a period and it is admissable for a mask or program name to contain more than one period.

Table 2-8: DOS ROM Functions

DOS Function 6Dh—ROM Find First

INPUT:

AH	6Dh
DS:DX	address of ASCIIZ mask to match with existing ROM program

SUCCESS:

cf	clear
DTA	filled with information describing first ROM program to match mask

FAILURE:

cf	set	
ax	0002h	if name is not found in ROM
	0003h	if name contains a colon or backslash

DOS starts its ROM scan at the segment recorded at the fixed address 0070:000A in the IO.SYS data segment. The search progresses upward, with DOS checking successive 2K boundaries. When a ROM module is found, its name tables are consulted in ascending order. If there is no match, then DOS resumes the search at the next 2K boundary beyond that ROM module. DOS does not check for ROM inside ranges given in the Exclusion List.

Table 2-8: Continued

DOS Function 6Eh—ROM Find Next

INPUT:

AH	6Eh
DTA	containing mask and resumption address

SUCCESS:

cf	clear
DTA	filled with information describing new match

Table 2-8: Continued

FAILURE:

cf	set
ax	0012h

The function copies the mask back over itself quite needlessly.

DOS Function 6F00h—Get Starting Segment for ROM Programs

INPUT:

AX	6F00h

OUTPUT:

cf	clear
al	00h
bx	starting segment for ROM modules containing DOS programs

DOS Function 6F01h—Set Starting Segment for ROM Programs

INPUT:

AX	6F01h
BX	starting segment for ROM modules containing DOS programs

OUTPUT:

cf	clear
al	00h

DOS Function 6F02h—Get Address of ROM Exclusion List

INPUT:

AX	6F02h

Table 2-8: Continued

OUTPUT:

cf	clear
al	00h
es:bx	address of Exclusion List

DOS Function 6F03h—Set Address of ROM Exclusion List

INPUT:

AX	6F03h
ES:BX	address of new Exclusion List

OUTPUT:

cf	clear
al	00h

If AL is out of range, function 6Fh sets the carry flag and returns 01h in AL.

Kernel Variations

Original Equipment Manufacturers (OEMs) redistribute DOS under their own labels. Although this may often amount to nothing more than rebadging, some OEMs make significant alterations, especially at lower levels and in utilities such as KEYB, that deal directly with hardware. Other variations arise when messages are translated into languages other than English and when double-byte character sets are supported.

In principle, OEMs distinguish their release from the generic DOS by assembling an identification number into MSDOS.SYS, from where it may be retrieved by software as part of DOS version-checking (see Table 2-9).

Table 2-9: DOS Function 30h—Get DOS Version

INPUT:

AH	30h
AL	00h for function to return OEM number in bh
	01h for function to return version flag in bh

Table 2-9: DOS Function 30h—Get DOS Version

OUTPUT:

ax	DOS version number—major version in low byte
bl:cx	user's serial number
bh	if AL ≠ 01h: OEM identification number if AL = 01h: version flag, in which bit 3 denotes the form of DOS that executes in ROM

The significance of AL when calling this function is new to DOS 5. In the present implementation all values of AL ≠ 01h direct the function to return the OEM number.

Although the official reference states that the version flag equals 08h for DOS in ROM, the DOSMGR virtual device in both Windows 3.0 and 3.1 actually identifies DOS in ROM by testing that bit 3 is set rather than comparing with 08h.

In DOS 5 and higher, the version number returned in ax is taken from offset 40h in the current PSP. Its value is set for version 5.00 at each PSP's creation, but may be adjusted by DOS function 4Bh (EXEC) after consulting a list of program names loaded into memory by the SETVER.EXE device driver. DOS 4 operates a similar scheme for deceiving programs about the version number but uses a global variable in the DOS kernel rather than maintain a separate version number for each process.

In versions before DOS 5 the number returned in ax is retrieved from a word of data in the DOS kernel. Curiously, DOS 5 retains this word at the same location (offset 0D12h) within the kernel's data segment as in version 4.01, despite making no use of it.

OEM Numbers

Some idea of the numbers assigned to OEMs may be obtained by studying the DOSMGR virtual device in Windows enhanced mode.

OEM NUMBER	OEM IDENTITY	3.10	3.20	3.21	3.30	3.31	4.00
00h	IBM	x	x	x	x		x
01h	Compaq	x	x			x	
02h	(default)	x	x	x	x	x	x
04h	AT&T		x				
05h	Zenith			x	x	x	
06h	Hewlett Packard		x				
23h	(unknown)	x	x				

Not all OEMs follow the practice of inserting an identification code into the information reported by DOS function 30h.

Table 2-10 below shows what impact the OEM version can have on the location of a variable within a single version of DOS. The table gives the offset within the IO.SYS segment of the byte in which the default CON device driver holds a waiting character, typically because a null byte is returned as the first component for a function key, Alt combination, and so forth, but also because Ctrl-Break has been entered between device driver calls.

Table 2-10: Example of Address Variation Within a Single DOS Version (3.20)

OEM NUMBER	SUBSIDIARY OR ALTERNATIVE CONDITION	ADDRESS
00h		0BC5h
FFh	also specific OEM numbers 02h, 03h, 07h to 23h	0BC4h
01h	or generic OEM number (00h or FFh), with the letters "COMPAQ" at 0070:0006	0B5Ch
04h	or generic OEM number, with the letters "AT&T Information Systems" at 0070:0005	0C47h
06h	or generic OEM number, with the letters "HP" at F000:00F8	0BD4h

The True DOS Version

Table 2-11: DOS Function 3306h—Get DOS Configuration and True Version (DOS 5 and higher)

INPUT:

AX	3306h

SUCCESS:

bx	DOS version number—major version in low byte	
dl	revision number	
dh	bitwise version flag	
	08h	denotes the form of DOS which executes in ROM
	10h	indicates that DOS is using the HMA, i.e., that **dos=high**

Official documentation limits the revision number to the lowest three bits returned in dl. This restriction is not recognized by the DOS 5 **ver /r** command, for which numbers 00h to 19h inclusive correspond to revisions A through Z, with higher values being reported as "Revision*".

Bit 4 is not set in dh until the DOS kernel's code segment has been moved to the HMA and prepared for execution there. In other words, the bit indicates that the **dos=high** configuration is complete rather than intended, a distinction which may be important to device drivers loaded before the XMS server.

Versions pre-dating DOS 5 do nothing but return FFh in register al, this being the usual behavior of int 21h function 33h when presented with an undefined subfunction.

Observe that in versions 4.0 and higher, int 21h functions 3303h and 3304h are transparent, not undefined. These subfunctions are genuinely reserved or obsolete, in the sense that they were set aside in an earlier version but now have no meaning, save only that new subfunctions must continue the sequence rather than reuse numbers 03h and 04h. It seems the two functions were intended to support a code page-switching flag along the same scheme as int 21h functions 3300h and 3301h to handle the break-checking flag. Although the DOS 4.01 kernel (at least the copy inspected) implemented the functions transparently, the SYSINIT module for that version admitted a **cpsw=on|off** statement in CONFIG.SYS for setting or clearing the flag with int 21h function 3304h. It is likely that this provision existed to support OS/2, which allows a code page to be set as a file attribute.

Kernel Variables

The SYSVARS Table

A DOS internal data structure, known variously as SYSVARS, SysInitVars, or the "List of Lists," contains information about the DOS kernel. The DOS SYSINIT module in IO.SYS uses this table (see Chapter 3 on the DOS startup sequence), hence the name SysInitVars. SYSVARS contains much information that is useful even after initialization and is crucial to almost all low-level DOS programming. Int 21h function 52h returns a pointer to this table in ES:BX.

Table 2-12: The DOS 5 System Variables (SYSVARS) Structure

OFFSET	SIZE	DESCRIPTION
- 02h	word	segment address of first memory arena header
00h	dword	address of first Drive Parameter Block (DPB)
04h	dword	address of first set of System File Tables (SFTs)
08h	dword	address of clock device driver

Table 2-12: Continued

OFFSET	SIZE	DESCRIPTION
0Ch	dword	address of keyboard device driver
10h	word	bytes in largest sector supported by any installed block device driver
12h	dword	DOS 4 and higher: address of **buffers** descriptor
	dword	DOS 3.30: address of first **buffer** in singly-linked list
16h	dword	address of first Current Directory Structure (CDS)
1Ah	dword	address of set of SFTs dedicated to FCB handling
1Eh	word	DOS 5: unused
	word	earlier versions: number of FCB-SFTs (counting from the most recently used) protected from pre-emptive closure
20h	byte	number of physical DOS drives; equivalently, number of DPBs
21h	byte	number of logical DOS drives; equivalently, number of CDSs
22h	12h	NUL device driver header; including pointer to most recently installed device driver
34h	byte	number of JOINed drives
35h	word	DOS 5: unused
	word	DOS 4: address of table (within kernel) containing names of programs and drivers requiring version number deception
	label	DOS 3.30: end of structure
37h	dword	DOS 5: address of table (typically supplied by SETVER) containing names of programs and drivers requiring version number deception
	dword	DOS 4: address of handler for IFS services
3Bh	word	DOS 5: only when **dos=high**—address within DOS kernel code of routine for fixing A20 control when executing special .COM format
	dword	DOS 4: address of first IFS header
3Dh	word	DOS 5: only when **dos=high**—PSP of process started most recently, for use in maintaining count of int 21h calls that disable the A20 on return
3Fh	word	number of disk **buffers**
41h	word	number of look ahead **buffers**
43h	byte	1-based number for boot drive
44h	byte	32-bit CPU flag; 00h if lower than 80386; else 01h for 80386 or better
45h	word	number of KB of extended memory available at system startup

Table 2-12 includes the word immediately before the structure because it is through this relationship that the word is always found, either by SYSINIT itself, by utilities such as MEM, or by the Windows DOSMGR device. There is however no suggestion that from DOS's perspective the structure extends any further backwards.

In DOS 3.30, the field at offset 12h addresses a disk buffer directly—the first in a singly-linked list. The descriptor structure which appears from DOS 4 onwards reflects the growing sophistication of the **buffers** system—particularly the EMS support in DOS 4 (but dropped from DOS 5) and the secondary (lookahead) buffering of consecutive logical sectors when reading at device driver level. The descriptor structure lies elsewhere in the DOS kernel's data area—at offset 006Dh. Although SYSINIT finds the descriptor by dereferencing the pointer at offset 12h in the SYSVARS structure, DOS itself does not. Thus, a program which installs a new set of **buffers** temporarily, say to test performance over a range of configurations, should not be tempted to replace both the buffers themselves and the descriptor (intending simply to redirect the SYSVARS pointer), but must instead take the more tedious approach of saving, modifying, and restoring the existing descriptor's contents.

The clock device driver whose address is recorded at offset 08h is the most recently installed character device driver with bit 3 set in its attribute word. The driver need not have the name CLOCK$, though DOS assumes that the driver reads and writes data in packets compatible with the default device driver of that name.

Similarly, the field at offset 0Ch records the address of the most recently loaded character device driver with bit 0 set in its attribute word. Again, the driver need not have the name CON. Indeed, it need not even be related to a keyboard. DOS's only interest in the device is for the detection of asynchronous breaks.

Some programs directly access the kernel data area, for example, by calling int 21h function 52h, ignoring the SYSVARS offset in BX, and then using hard-wired offsets into the segment returned in ES. Chapter 13 discusses the example of MSCDEX.

Windows Support

Despite the assertion in Chapter 1 that the construction of protected-mode Windows on top of real mode DOS is beyond the scope of this book, it is clearly a recurring topic. A discussion of the DOS kernel necessarily involves an examination of how DOS 5 and higher supports Windows.

Windows uses int 2Fh to issue several broadcasts when starting up and exiting—see Table 2-13. DOS 5 and higher responds to these broadcasts.

Windows aborts its initialization if the broadcast fails. User-level explanation, if any, must be delivered by the client responsible for the failure. Windows standard mode concerns itself with nothing other than the success or failure of the function; in other words, it uses the broadcast only to determine whether some client wants to prevent Windows from running.

Table 2-13: Windows Startup Broadcast—Int 2Fh Function 1605h

INPUT:

AX	1605h
DX	bit 0 set for standard mode
DI	Windows version number, major version in high byte
CX	0000h
DS:SI	0000:0000
ES:BX	0000:0000

SUCCESS:

cx	0000h
ds:si	for enhanced mode: address of memory manager calling point to disable virtual-8086 (V86) mode
es:bx	for enhanced mode: address of structure giving startup information

FAILURE:

cx	non-zero
ds:si	corrupt
es:bx	corrupt

Windows does not rely on clients to preserve registers (and Windows standard mode assumes explicitly that registers DS and ES may be corrupted).

Clients may use the broadcast, for example, solely to track Windows' activity for their own purposes. Software loaded before Windows (such as TSRs and DOS device drivers) can maintain a flag indicating whether Windows is currently running. They will likely also want to return information, in which case they should first pass the broadcast (unchanged) to the previously-installed int 2Fh handler and only afterward provide their own information. DOS 5 and higher uses the broadcast for both purposes: to track when Windows enhanced mode is running and to return information to Windows during its initialization.

The memory manager calling point is an address that Windows must obtain from pre-existing protected-mode software in order for Windows to take control of the processor. When Windows calls the function, the memory manager attempts to switch from V86 mode back to real mode.

Each startup information structure has the form:

OFFSET	SIZE	DESCRIPTION
00h	word	Windows version, major version in low byte
02h	dword	far pointer to next startup information structure
06h	dword	far pointer to ASCIIZ pathname for virtual device driver
0Ah	dword	reference data for virtual device driver
0Eh	dword	far pointer to array of instance data structures

Clients use the far pointer at offset 2 to maintain a chain of these startup information structures.

Each instance data structure describes a region of memory in V86 address space whose contents must be maintained separately for each VM:

OFFSET	SIZE	DESCRIPTION
00h	dword	linear address of region's base (0 to mark end of array)
04h	word	size of region in bytes

Table 2-14: Windows Exit Broadcast—Int 2Fh Function 1606h

INPUT:

AX	1606h
DX	bit 0 set for standard mode

Windows enhanced mode is not interested in any values returned by this function and assumes nothing about register preservation. Windows standard mode assumes that DS is preserved.

Under certain circumstances (a fatal exit), Windows neglects to issue the 1606h exit broadcast.

As noted earlier in the discussion of BLOCKDEV (see Table 2-4), individual VxDs can use int 2Fh function 1607h as a broadcast to software (such as DOS) loaded before Windows; the VxD places its ID number in BX. For Windows enhanced mode's interaction with MS-DOS, a key VxD to consider is, of course, DOSMGR, whose VxD ID is 15h. The DOSMGR function 1607h broadcast is handled by MS-DOS 5 and higher, as shown in Table 2-15.

Table 2-15: DOS Kernel Support for Windows via Int 2Fh Functions (DOSMGR Callouts)

DOSMGR DEVICE BROADCAST 00H—ESTABLISH DOSMGR SUPPORT

INPUT:

AX	1607h
BX	0015h
CX	0000h
DX	0000h

SUCCESS:

cx	non-zero—DOS 5 returns 0001h
dx	segment address of IO.SYS data; defaults to 0070h if dx is not changed from zero
es:bx	address of table containing offsets within DOS kernel of certain variables:

OFFSET	SIZE	DESCRIPTION
00h	word	DOS version—major version in low byte, e.g., 0005h for DOS 5.00
02h	word	address of variable used to store register ds temporarily during int 21h dispatch
04h	word	address of variable used to store register bx temporarily during int 21h dispatch
06h	word	address of InDOS flag
08h	word	address of variable containing machine ID
0Ah	word	address of null-terminated array of words pointing to patch bytes for routines that signal critical sections
0Ch	word	expected only by Windows 3.1 and regarded as meaningful only for DOS 5 onward: address of word in which DOS records segment address of last arena header in conventional memory (see DOS function 5803h in Chapter 4)

Supporting this function implies that instance data for the two system files is available via the Windows initialization broadcast (int 2Fh function 1605h). Instance data for IO.SYS should include the **stacks**, if configured. Instance data for MSDOS.SYS should not include the Swappable Data Area, which DOSMGR (like other programs) is perfectly well able to locate for itself (and determine the InDOS and swap-always sizes of) via undocumented int 21h function 5D06h (Get SDA).

If Windows does not receive a positive response from this function, it attempts to find instance data through int 2Fh function 1603h (see Table 2-16 below). Failing this, Windows falls back on either the INSTANCE.386 text file or tables embedded in the DOSMGR device. By default Windows consults its own tables first, but this order of preference may be overridden by supplying the Boolean keyword **UseInstFile** in the **[386Enh]** section of SYSTEM.INI. The internal tables cover DOS versions 3.10 to 4.00 inclusive and cater for known IO.SYS variations introduced by OEMs.

Table 2-16: DOSMGR Callouts, Continued

DOSMGR Device Broadcast 01h—Enable Specialized Windows Support

INPUT:

AX	1607h	
BX	0015h	
CX	0001h	
DX	bit flags describing required support:	
	0001h	enable critical section signals (int 2Ah functions 80h and 81h)—to support re-entering DOS while InDOS flag is non-zero
	0002h	allow non-zero ID for local machine—to separate VMs using network mechanism
	0004h	decompose binary reads from devices in console mode—to increase frequency of int 2Ah function 84h scheduling
	0008h	arrange notification of system halt due to Internal Stack Failure
	0010h	arrange notification of imminent change in logical drive map

SUCCESS:

dx:ax	A2AB:B97Ch
bx	bit flags from DX; bits are cleared if the corresponding form of support is not installed

Windows attempts to enable all these forms of support by its own means unless this function returns the expected signature in dx:ax. Windows does this by searching the DOS kernel (flags 1, 2, 4) or IO.SYS (flags 8, 10h) for sequences of relevant instructions that indicate positions for run-time patches or V86 breakpoints.

Some of the specific instruction sequences involved in the searches contain references to data in the DOS kernel. If possible, these references are resolved through the table supplied at es:bx by DOSMGR broadcast 00h (Table 2-15).

Without that table DOSMGR defaults its kernel references to values expected for DOS 3.xx, unless the DOS version obtained from int 21h function 30h is 4.00. Note the implication: For DOS versions higher than 4, failure to support the DOSMGR broadcasts (for example, a DOS clone reporting version 5.0 but failing to hook int 2Fh and handle the DOSMGR function 1607h broadcast) would leave Windows searching for instruction sequences based on DOS 3.xx addresses for the relevant kernel data. Failure to find sequences involved for the DOS kernel patches is fatal to Windows initialization.

In the special case of DOS in ROM it would be pointless for Windows to try overwriting system code, so only the critical section patches are installed. In versions before DOS 5 and not designed to execute in ROM, each routine for signalling critical sections has its own patch byte in the kernel code—to be changed from a RET instruction (which trivializes the routine) to a PUSH AX. DOS 5 and versions of DOS in ROM set aside a single patch byte in the data segment; each routine consults this byte in order to decide whether to act trivially or to signal a critical section. For compatibility with DOS 3.x, the critical-section patch array still exists, although with all entries pointing to the same byte.

DOS 5 and higher are aware of Windows' requirements and simply return the flags unchanged. The necessary support is enabled on detecting the Windows initialization broadcast (int 2Fh function 1605h) for enhanced mode.

Table 2-16: Continued

DOSMGR Device Broadcast 02h—Disable Specialized Windows Support

INPUT:

AX	1607h
BX	0015h
CX	0002h
DX	bit flags describing services to disable

No return values are consulted. DOS 5 and 6 return cx = 0000h. Windows issues this function when terminating the System Virtual Machine, but only if Broadcast 01h was supported during DOSMGR Device Initialization.

Table 2-16: Continued

DOSMGR Device Broadcast 03h—Get System Structure Sizes

INPUT:

AX	1607h
BX	0015h

Table 2-16: Continued

CX	0003h
DX	0001h

OUTPUT:

dx:ax	A2AB:B97Ch
cx	size of Current Directory Structure (CDS) in bytes

DOS 5 and 6 return cx = 0000h unless bit 0 is set in DX. The DOSMGR device issues this function for no purpose other than to obtain the CDS size, but a reasonable conjecture is that the interface may have been, or may still be, intended more generally.

One obvious reason why Windows needs to know the CDS size is that subject to the behavior of the next function it instances the whole CDS array to allow each virtual machine its own relationship between drives and directories.

The other reason, far from obvious, is to help Windows instance the **stacks** when it has not already been passed their location and size through other instancing interfaces such as int 2Fh function 1605h. In early DOS versions, the stack pool is handled within IO.SYS, which provides a structure containing the location, size, and number of stacks. From DOS 3.30 onward, stack-sharing is more configurable and the stack pool is built in the same block of memory used for device drivers, SFTs, the **buffers** and the CDS array. Specifically, if Windows has to locate the stack pool by its own means it assumes that the code and data required for the **stacks** follows the CDS array. Note what this imposes on those programs (some of which are many years older than the 80386 development of Windows) that relocate the CDS array to upper memory. Perhaps some memory managers recommend **stacks=0,0** to avoid not having the stacks be instanced.

Table 2-16: Continued

DOSMGR Device Broadcast 04h—Get Instancing Exemptions

INPUT:

AX	1607h
BX	0015h
CX	0004h

SUCCESS:

dx:ax	A2AB:B97Ch

Table 2-16: Continued

bx bit flags denoting typically external data structures that Windows need not instance:

0001h	do not instance Current Directory Structures
0002h	do not instance chain of System File Tables or device status of stdout
0004h	do not instance device driver chain
0008h	do not instance the DOS kernel's Swappable Data Area

DOS 5 and 6 fail this function by returning zero in both cx and dx.

DOSMGR Device Broadcast 05h—Get Size of Device Driver

INPUT:

AX	1607h
BX	0015h
CX	0005h
ES:DI	address of device driver header

SUCCESS:

dx:ax	A2AB:B97Ch
bx:cx	size of device driver in bytes

DOSMGR issues this function when instancing a device, typically because the **[386Enh]** section of SYSTEM.INI contains a **Local** entry for the device's logical name. Each virtual machine receives its own copy of the *whole* device driver.

For this function to succeed in DOS 5 and 6, the device driver header must be preceded by a memory arena header with the 'D' signature—as will normally be constructed under DOS versions 4 and higher when device drivers are loaded through CONFIG.SYS. If this is not the case, DOS 5 and 6 fail the function by returning zero in ax, cx, and dx.

Windows 3.0 Real Mode

The DOS 5 kernel responds to the Windows 3.0 task switching interface. Windows 3.0 real mode has a problem with the way that dos=umb causes the ceiling segment on conventional memory to be odd; the sentinel at the end of the real-mode Windows heap is aligned on an even paragraph, expecting to leave one paragraph spare, that is, a sentinel at 9FFEh so that 9FFFh is spare. DOS 5 therefore intercepts int 2Fh functions 4601h d 4602h (issued by WINOLDAP when switching) in order to save the real-mode ntinel from corruption.

Vindows/386

hile seemingly only of historical interest, Windows 3.x will still issue the otherwise-bsolete Windows/386 function call to get instance data (int 2Fh function 1603h), if ther ways of getting information about instance data have failed. For example, if a DOS one reports version 5.0 but fails to handle the int 2Fh function 1605h broadcast, indows enhanced mode will try the old Windows/386 broadcast, shown in Table 2-17.

---

able 2-17: Int 2Fh Function 1603h—Get Windows/386 Instance Data

NPUT:

X 1603h

UCCESS:

x 5248h

s:si address of instance data in Windows/386 format

OFFSET	SIZE	DESCRIPTION
00h	word	segment address of IO.SYS; or 0000h to accept 0070h as default
02h	word	offset within IO.SYS of structure describing the **stacks** configuration in DOS 3.20 and 3.21; or 0000h if not applicable
04h	word	number of instance data structures
06h		array of instance data structures

Windows accepts no more than 32 instance data entries via this function. These have e form:

FFSET	SIZE	DESCRIPTION
0h	word	segment; or 0002h to denote the DOS kernel
2h	word	offset
4h	word	size

hich corresponds directly to the text entries in the INSTANCE.386 file. This file may ontain descriptions for more than one DOS version, under headings such as **[DOS .20]**. The general form in which regions of instance data are described is a gment:offset address followed by the size in bytes. Again, segment 0002h denotes the OS kernel. To complete the correspondence, observe that INSTANCE.386 allows a special entry containing the value 1 followed by the IO.SYS segment (or zero for the default segment 0070h) and the offset of the **stacks** header.

Driving Windows from DOS

VxDs can provide APIs to Windows programs or to DOS programs running under Windows. Software loaded before Windows (such as DOS itself) can also call into VxDs via these APIs. As shown in Table 2-18 below, int 2Fh function 1684h returns the address of the API entry point for a specified VxD. Calling int 2Fh function 1684h in V86 mode will return a V86 API entry point, if available.

This is relevant to DOS internals because MS-DOS 5 and higher call into the DOSMGR VxD, using its V86 API returned from int 2Fh function 1684h.

Table 2-18: Locating a Virtual Device's API with Int 2Fh

INPUT:

AX	1684h
BX	VxD ID, e.g., 0015h for DOSMGR

SUCCESS:

es:di	address at which to call device's API, or 0:0 if no API provided for current processor mode (V86 API if called in V86 mode, PM API if called in protected mode)

FAILURE:

es:di	0000:0000

When virtual-8086 software issues this request, it is detected and answered by the VMM device in ring 0. Note especially that virtual-8086 software cannot hook int 2Fh in order to return a different calling address for the device API.

Not all devices offer a V86 API, hence the possible failure of the function. In circumstances where it is not certain that Windows enhanced mode is already running (for example by calling int 2Fh function 1600h), it would be wise to clear ES:DI before calling the function.

If a V86 API is available from the nominated device, then the first attempt to obtain the calling address is dealt with by allocating a V86 Call Back—a unique far pointer to the reusable Windows breakpoint (which may lie in either the Global V86 Data Area or in the System ROM). VMM stores the far pointer in the device's descriptor block for future reference.

While the int 2Fh function 1684h mechanism is documented in the Windows DDK, the APIs provided by specified VxDs tend not to be. DOSMGR's V86 API provides functions for VM control, int 2Ah (critical section) modification, and for media-change detection. For example, under certain circumstances IO.SYS calls DOSMGR function 1 (Set Critical Focus).

Table 2-19: V86 API Provided by DOSMGR and Accessed by Far Call

DOSMGR V86 API FUNCTION 0000H—GET DOSMGR VERSION

INPUT:

AX	0000h

Output:

cf	clear
ax	DOSMGR version number—major version in high byte

The DOSMGR version (which will ordinarily, but not necessarily, equal the Windows version) determines which of the remaining API functions are supported. Version 3.0 recognizes only the functions up to and including 0002h.

Table 2-19: Continued

DOSMGR V86 API Function 0001h—Set Critical Focus

INPUT:

AX	0001h

OUTPUT:

cf	clear

DOSMGR V86 API Function 0002h—Crash Current VM

INPUT:

AX	0002h

There is no return from this function. Windows 3.1 precedes the VM's destruction by recording a Detailed VM Error, so that a message box may inform the user that the "application has been stopped by the DOSMGR device."

Table 2-19: Continued

DOSMGR V86 API Function 0003h—Enter Critical Section

INPUT:

AX	0003h

This function is private to the code which modifies int 2Ah functions 8001h and 8002h issued by the DOS kernel and other system components such as SHARE. It is taken for granted that the int 2Ah modification has been prepared.

Table 2-19: Continued

DOSMGR V86 API Function 0004h—Get VM ID Byte

INPUT:

AX	0004h

SUCCESS:

cf	clear
es:di	address of byte containing the VM ID

FAILURE:

cf	set

The function fails if the int 2Ah modifications have not been prepared. Virtual-8086 software may obtain the ID of the current VM via int 2Fh function 1683h, which returns the ID in register bx and clears al to zero. It may solve some problems to allow determination of the VM ID by simple inspection of memory at an address that is the same in all VMs.

Table 2-19: Continued

DOSMGR V86 API Function 0005h—Inform Windows of Possible Media Change

INPUT:

AX	0005h
BL	0-based drive number

Table 2-19: Continued

SUCCESS:

cf	clear

FAILURE:

cf	set

This function does nothing if **RemMediaCDChngTrack=off** in the **[386Enh]** section of SYSTEM.INI. The default setting is on. See the earlier discussion of disk changes.

The Command Processor

In version 3.30, COMMAND.COM is well known to consist of three parts. The first part (shaded in Figure 2-4) stays in memory until the command processor terminates. The second portion is discardable initialization code. The remainder of the file—by far the largest part—is a transient component that generates the command line, deals with internal commands such as **copy** and **dir**, executes child programs, and processes batch files. The transient component is separately addressable from the other sections to support its relocation away from the resident portion before any child programs are executed.

From DOS 4 onward, COMMAND.COM has contained default error messages accessible to software via an undocumented interface. Details of the interface are given in Chapter 17 under the heading "Error Code Translation," but for present purposes, the important point is that these messages are kept in memory only by the top-level instance of COMMAND.COM and only if started with the **/msg** switch. This area of conditionally-retained memory is shown in light shading.

With or without the error messages, the memory footprint nearly doubled in version 4. DOS 5 offers two remedies. As shown in Figure 2-4, the resident section is split. As with the main system files, a large piece of resident material (the second darkly shaded portion) goes to the HMA if **dos=high** is selected. Additionally, this relocatable piece is shared code. When multiple instances of the command processor are run, only the one copy of this section is kept in memory. Even on a lowly 8086, this saves just 2K whenever the user shells out from one application to run a DOS command—not much perhaps, but it's still a nice touch on these old machines, for which not much else can be done. From a programming perspective, the feature is significant because it more or less obliges COMMAND.COM to provide (for the first time) an interface through which the top-level instance may be found—see Table 2-20.

Figure 2-4: Structure of COMMAND.COM on Disk—Comparison of Different DOS Versions

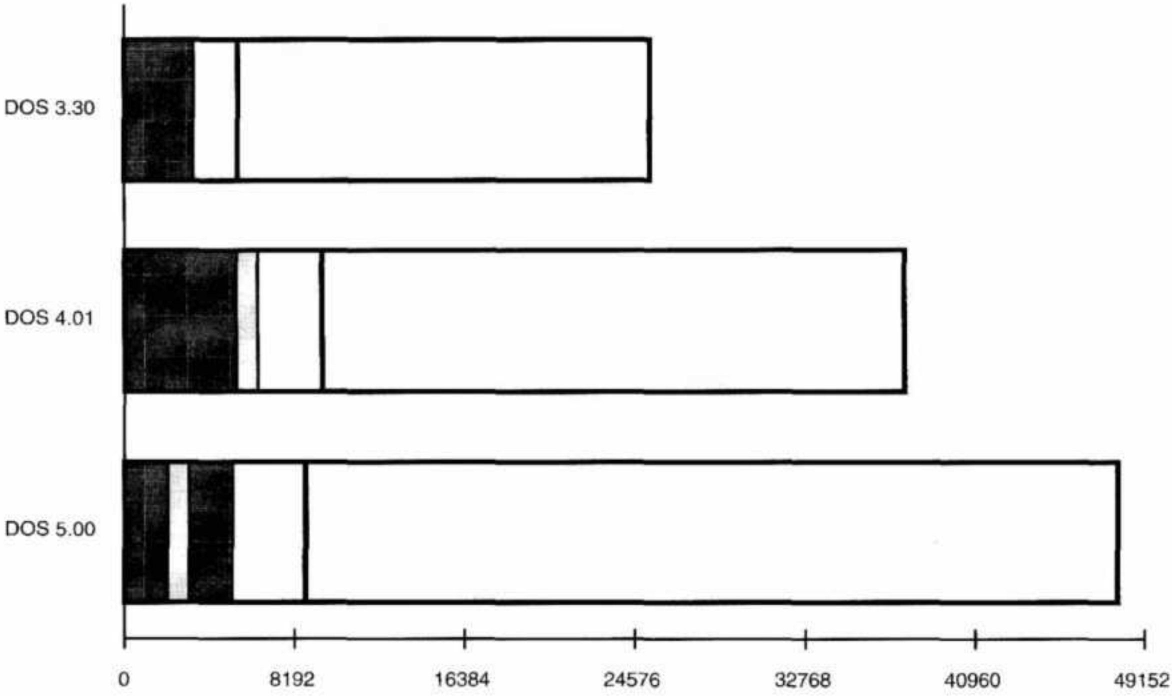


Table 2-20: Int 2Fh Support for Multiple Copies of COMMAND.COM in DOS 5

INPUT:

AH	55h	
AL	00h	for disk-based version
	01h	for ROM-executable version

OUTPUT:

ax	0000h
ds:si	address of table in the top-level instance of COMMAND.COM—the table is an array of far pointers to code in the segment shared among multiple instances

The int 2Fh function 55h interface is installed by the first copy of COMMAND.COM that is started with the **/p** switch (with the exception that COMMAND.COM disregards **/p** if **/c** is also present). Subsequent instances of COMMAND.COM copy the table over the corresponding space in their own data segment—gaining access to the shared code without needing further communication with the resident component of the top-level processor.

The addresses in the table should be regarded as private to COMMAND.COM. They are the entry points to the HMA- or ROM-executable portions of larger routines, some of which have well-defined, publicly accessible entry points in low memory—interrupts 23h, 24h, 2Eh, and 2Fh and also an error message translator reached by far call.

A small defect in this scheme (or at least in the way it is presently implemented) is that if **dos=high**, then the shared segment is copied to the HMA when COMMAND.COM thinks it is being run for the first time, but int 2Fh function 5500h is left behind only if COMMAND.COM is started with **/p**. Thus, until COMMAND.COM is run once with the **/p** switch, each instance will believe itself to be the first and will copy its shared segment to the HMA, apparently without realizing that it will not be installing the int 2Fh interface through which subsequent instances may locate the "shared" material. A practical consequence is the possibility when **dos=high** of wasting the HMA by including sufficiently many CONFIG.SYS statements of the form

```
install [[drive:][path]]command.com /c [loadhigh] program [arguments]
```

Each such line causes 2K or so of COMMAND.COM code to be appended irreversibly to the system material already placed in the HMA (remember that the int 2Fh function 4Ah interface through which IO.SYS apportions the HMA among DOS components does not provide for deallocation). The example features the internal command **loadhigh** to lend the unusual construction some faint plausibility as an attempt to create an **installhigh** facility (admittedly at the price of the usual error message generated when loading a non-resident program with **install**).

CHAPTER 3

The Startup Sequence

Preparation for DOS's possible move to the HMA has brought not only an internal reorganization of the system files, but also a rearrangement of the startup sequence. The dynamics of system configuration have to accommodate the four regimes into which DOS memory management must settle by the time the shell is ready to run.

Juggling the System

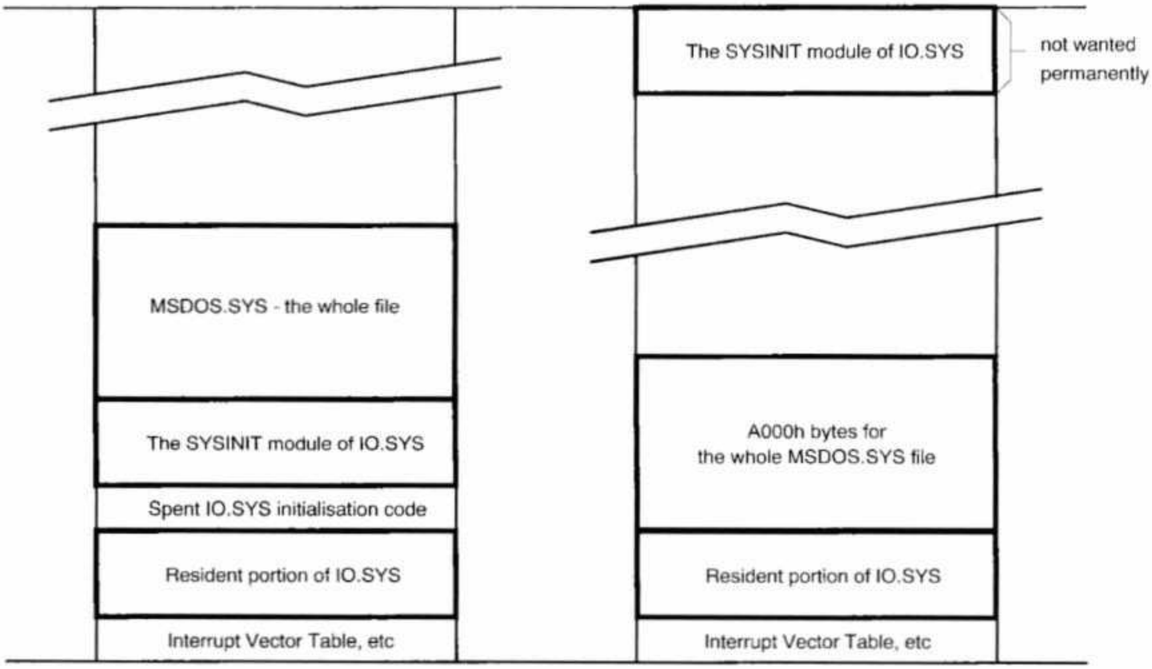
Consider that **dos=low,noumb** is the base state. It requires the least effort to configure and is the least powerful operating regime. It is the state into which the system falls if nothing special is attempted. At the other extreme, **dos=high,umb** is a stable state with a high potential, but a narrow basin of attraction. There is greater promise in this regime, but the system can reach this state only if a number of conditions are satisfied. An XMS server and a UMB provider are both required, but will not be available at least until the system is ready for installable device drivers. It is even conceivable that one or the other might exist only in the form of a normal executable, unable to be loaded until the system has progressed far enough along its startup sequence that it can run programs. For all this time the system must be held in a suspended state capable of settling into the desired regime the moment that all appropriate conditions are met, but equally able to shift safely to **dos=low,noumb** if it becomes clear that the desired regime is not achievable for example because an XMS server or UMB provider is not in fact installed.

Before DOS 5

This **dos=low,noumb** base state is roughly comparable to earlier DOS versions. If only to secure a point of reference before proceeding with the complex preparations required by DOS 5 to arrange a choice of states, it may be best to see how these early DOS versions set their stage. Immediately after IO.SYS finishes its initialization, having just loaded the MSDOS.SYS file from disk, the layout in memory is something like the left side of Figure 3-1. SYSINIT's first task is to move itself out of the way and to move the MSDOS.SYS image downward, as shown on the right-hand side of Figure 3-1.

The relocations produce a compact image which the DOS kernel reduces a little more when it sheds its initialization code. From its position high in memory, SYSINIT loads device drivers, builds system structures, and possibly also installs resident programs—stacking them all on top of the system in low memory. Thus, when the transient SYSINIT module finishes its work and starts a command processor, it leaves no "hole" (an isolated piece of free memory too small to be generally useful).

Figure 3-1: Before and After Relocation of SYSINIT in Versions Earlier than DOS 5

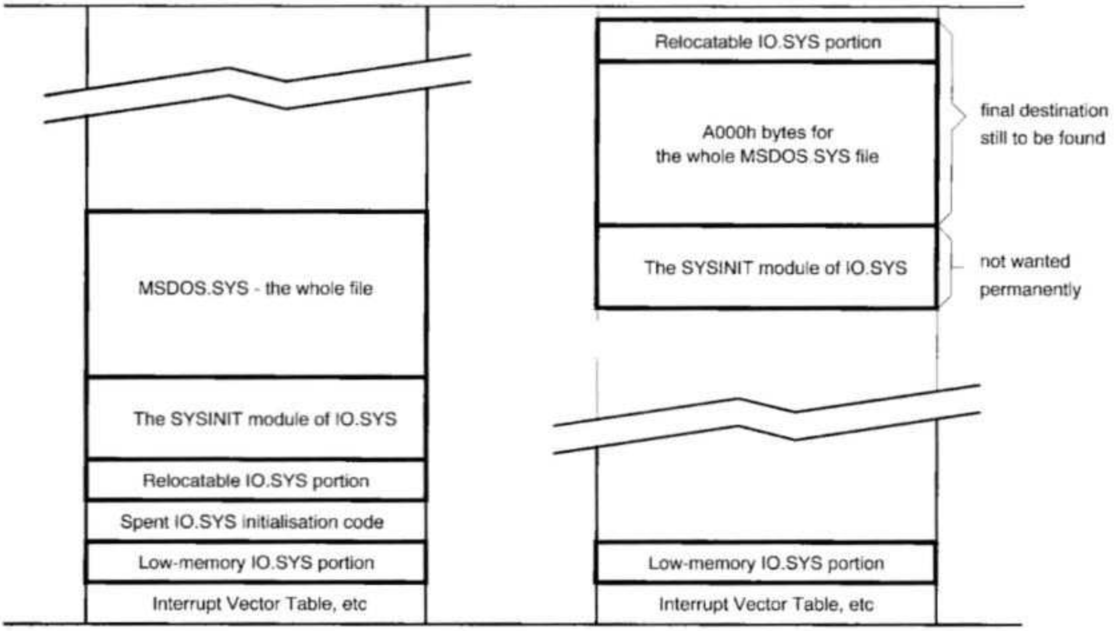

The SYSINIT module of IO.SYS
not wanted permanently
MSDOS.SYS - the whole file
The SYSINIT module of IO.SYS
A000h bytes for the whole MSDOS.SYS file
Spent IO.SYS initialisation code
Resident portion of IO.SYS
Resident portion of IO.SYS
Interrupt Vector Table, etc
Interrupt Vector Table, etc

DOS 5

The corresponding picture for DOS 5 (presented in Figure 3-2) is complicated a little by the existence of more pieces. Again, the left side of the diagram shows the layout immediately after IO.SYS initialization. All the system components are low in memory. Instead of one resident IO.SYS component, however, there are two—a data segment which is already in its expected place (at segment 0070h) and a code segment whose ultimate position depends on whether a **dos=high** regime is both wanted and achievable.

It is with this uncertainty over final destination that the complications really begin. As already stated, the system must be held in a suspended state pending resolution of the operating regime. At this very early stage in the system's initialization nothing is known about the user's choice of configuration, let alone whether this choice can be realized. The immediate consequence is seen on the right-hand side of Figure 3-2, which shows how almost all of the system is held temporarily near the top of conventional memory.

Figure 3-2: Before and After Relocation of SYSINIT in DOS 5


Relocatable IO.SYS portion
A000h bytes for the whole MSDOS.SYS file
final destination still to be found
The SYSINIT module of IO.SYS
not wanted permanently
MSDOS.SYS - the whole file
The SYSINIT module of IO.SYS
Relocatable IO.SYS portion
Spent IO.SYS initialisation code
Low-memory IO.SYS portion
Interrupt Vector Table, etc
Low-memory IO.SYS portion
Interrupt Vector Table, etc

In the default **dos=low** regime, all resident system modules will go to the bottom of memory as before; indeed, this is arranged as soon as the user's choice of this configuration is discovered. On the other hand, if the user selects **dos=high**, then the final resting places for the resident system modules may not be known for some time—at least not until an XMS server has been loaded, probably as a device driver. Until then the relocatable system components have to be kept out of the way so that non-system software destined for low memory may be packed into place as efficiently as possible. Note the need for a "Plan B" in the **dos=high** case. It may turn out that SYSINIT cannot claim the HMA (most extremely, because no XMS server is ever loaded) and that the system components must end up in low memory after all.

In a **dos=umb** regime, the chain of memory arena headers with which DOS keeps track of memory will extend beyond the memory reported by the ROM BIOS, necessitating an allowance for one paragraph at the top of memory now, just in case the space is needed for an arena header later. Of course, the paragraph will not be needed if it turns out that there are no UMBs by the time the startup sequence draws to its end. The point is that preparation must begin now. Until CONFIG.SYS has been analyzed, SYSINIT cannot determine if UMB management will be required (let alone whether upper memory will ever exist). It should also be clear that determining the **dos** setting in CONFIG.SYS must be one of SYSINIT's highest priorities.

Allowing for system objects to be moved around pending a choice of final destination brings with it a host of practical difficulties. At this early stage, the MSDOS.SYS file can be moved at will. It is completely inert since it has not been called to initialize itself. This is certainly not true for the various pieces of IO.SYS. Both the low-memory and the relocatable components have executed at least some code and each may have stored information about the other's location (and about SYSINIT also). Moving these modules therefore involves a little inter-modular communication to make sure that far pointers are kept up-to-date. This is easy enough, since the pieces being juggled are all from the same file and the addresses of procedures that one module needs to call in another will have been built-in at link-time. It won't be so easy later, when both system files are involved. More and more information is built into the system as the startup sequence progresses, so the eventual relocation of system modules becomes increasingly problematic the longer it is delayed.

Headroom

Another way in which suspending the system stores trouble for later is that device drivers may be troubled by the significant increase in the amount of system material floating about near the top of memory. This is not just a concern for low-memory situations that many may be tempted to dismiss as having nothing more than historical interest (after all, how many machines nowadays have so little base memory that device drivers could conceivably overrun the suspended system modules?).

No, lowering the effective ceiling on memory available for device driver initialization causes much distress to PROTMAN.SYS, the protocol manager for Microsoft-compatible networks. Like the system files, this driver must find a temporary place to stay in memory, waiting for the right conditions, in this case until remaining network components are loaded and ready for binding. Past versions of PROTMAN find the top of conventional memory, then subtract 32K to obtain an address that it presumes may be used. Until DOS 5, this may have been a reasonable allowance for the system, but not anymore (and matters get worse with DOS 6, which adds DBLSPACE.BIN to the list of modules that may hang near the top of memory for a while). Whether or not PROTMAN's designers could have found some less arbitrary way to assess the amount of temporary system material than just thinking of the number 32 is not really the point (just now). The choice faced by DOS's designers is that either PROTMAN must be replaced with all DOS upgrades or that DOS 5 had to find some way to make PROTMAN's behaviour safe. In the best tradition of software kludging, it turns out that whenever DOS 5 notices it is about to load a device driver with the logical device name PROTMAN$, it revises the BIOS memory count (the word at 0040:0013) far enough downward that PROTMAN may go ahead with its allowance of 32K as before, but from an address that is already below the system modules. Once the PROTMAN$ device has initialized, SYSINIT restores the BIOS memory count.

Activating the DOS Kernel

Having sorted itself out as best it can, SYSINIT calls the DOS kernel. The parameter passing convention used for MSDOS.SYS initialization is far from useful in any practical sense, but is an invaluable aid to understanding the startup environment, because it describes almost everything that SYSINIT and the DOS kernel know about each other. The parameters are shown in Table 3-1.

Table 3-1: Parameters for SYSINIT's Far Call to Initialize MSDOS.SYS (DOS 3.30 and Higher)

INPUT:

DX	ceiling segment for conventional memory
DS:SI	address of default CON device driver
ES:DI	address of InConfig flag—new to DOS 5

OUTPUT:

ax	bytes to retain if **dos=high**—new to DOS 5
cx	bytes to retain if **dos=low**—new to DOS 5
dx	address to notify kernel about relocation—new to DOS 5
ds	segment address of a newly-created PSP
es:di	address of array of far pointers to objects in the kernel's data segment:

OFFSET	SIZE	DESCRIPTION
00h	dword	address of kernel's System Variables (SYSVARS) structure
04h	dword	address of kernel's Country structure

In early DOS versions, this exchange of parameters is just enough that the kernel may learn the results of IO.SYS initialization and that SYSINIT knows where to find the kernel structures that must be modified to fit the choice of configuration. Remember that in these versions, the whole MSDOS.SYS image is already in place before initialization—as shown in Figure 3-1. SYSINIT need know nothing of the kernel's internal organization, save only that the initialization code must be called at offset 0000h in the file's image. Note especially that the kernel is wholly in control of shedding any part of its image dedicated to initialisation.

The additional parameters returned by the DOS 5 kernel in Table 3-1 correspond to the broad features depicted graphically in the Chapter 2's diagram of the MSDOS.SYS layout. The DOS 5 kernel has more structure, seen especially in terms of its separate segmentation (and reversal of ordering) for code and data. As with earlier DOS versions, SYSINIT has no prior knowledge of this internal organization. It is one thing for SYSINIT to know about its companion modules within IO.SYS, but the DOS kernel is a separate file. There are limits to how much one system file should know about the other without having to channel the information through an interface at run-time.

Moving Kernel Data to Low Memory

As with IO.SYS, the MSDOS.SYS data must lie in low memory, whether **dos=high** or not, since it contains semi-public data such as the CDS, SFT, and so on, that must always be available for direct inspection. Moreover, the data segment must be in its final place before any user-supplied software is loaded. Interfaces such as int 21h function 52h return addresses of kernel data structures, and software is unlikely to react well to having these addresses invalidated.

Since SYSINIT is unable to differentiate code from data within the MSDOS.SYS image, the early relocation of kernel data to low memory must be performed as part of MSDOS.SYS initialization—indeed, as the very first task. The natural destination for the kernel data segment is immediately after its IO.SYS counterpart at 0070h. The required address is not passed as an input parameter to the MSDOS.SYS initialization routine. Instead, DOS inspects the word at 0070:0003, knowing that this is where the long-gone IO.SYS initialization code records the segment address left free by the IO.SYS data module (whose footprint depends on things like the number of hard disk partitions). The observation that this word doubles as the segment address of MSDOS.SYS data may serve as a handy shortcut in debugging sessions on an unknown computer. Readers in the habit of inspecting kernel data will know the relevant segment for their own machines already.

While on the subject of specific addresses set aside by the system in DOS 5, note that MSDOS.SYS also understands that 0070:0005 is the entry point to the IO.SYS component of the system files' joint int 2Fh handler. When the kernel passes int 2Fh to IO.SYS, it jumps to the address 0070:0005. Of course, IO.SYS does not have its entire int 2Fh handler at this address but instead diverts execution to somewhere more convenient. It happens that IO.SYS does this by a far jump to another location in the IO.SYS data segment, where yet another far jump sends control to the IO.SYS code segment (which might be located in the HMA). The main point here is that the far jump instruction at 0070:0005 in both DOS 5 and DOS 6 is potentially patchable so developers of anti-virus software may as well start their defensive preparations now.

Planning Ahead for Moving Kernel Code

It should now be clear why DOS 5 reverses the order of code and data in the MSDOS.SYS file. The new arrangement puts the discardable initialization code and the whole data segment together at the end of the file. Neither of these can possibly concern SYSINIT once the kernel has been initialized. The first will have played its part already and the second will be safely at its final destination in low memory where it requires no further attention. Thus, SYSINIT need know only how many bytes of the MSDOS.SYS image in high memory must eventually be moved either to low memory or to the HMA. Observe that the interface allows the kernel to declare two sizes (in registers ax and cx). A little extra code must be retained if DOS goes to the HMA to deal with problems that some programs encounter when DOS keeps the A20 more or less permanently enabled.

As confirmation that module relocation is becoming increasingly complicated, see in Table 3-1 that SYSINIT is given the address of a routine (in register dx) through which the kernel must be kept up-to-date about its changing circumstances. The various IO.SYS modules have corresponding routines, but knowledge of their locations is built in at link-time.

Marking Progress

As well as returning additional parameters to SYSINIT, the DOS 5 kernel also expects an extra piece of information for its initialization. The address passed in ES:DI is for a byte in the IO.SYS data segment. This InConfig flag (for want of a better name) is turned on (01h) when SYSINIT has moved itself to the top of memory and then back off (00h) once SYSINIT has completed its configuration of system structures and is ready to **install** programs.

This demarcation is already familiar from the discussion of int 2Fh functions 4A01h and 4A02h in Chapter 2. Remember that these functions comprise the interface for apportioning the HMA among DOS components when **dos=high**. If the IO.SYS and MSDOS.SYS code segments are not yet in the HMA when either function is called during a device driver's initialization, that is, while the flag is still set, then SYSINIT is directed to perform the relocation straightaway, and only then does the IO.SYS code proceed with the function. It is not clear just why this ruse is abandoned once SYSINIT has progressed to running programs, but it happens because the IO.SYS int 2Fh handler monitors this InConfig flag.

The DOS kernel checks the flag to distinguish when int 21h function 4B03h is being used to load device drivers rather than program overlays. Only in the former case will DOS consult the SETVER table to see if version number deception is required. Presumably the argument proceeds along the line that device drivers have a high profile but overlays are not generally considered programs in their own right. Note the difficulty this poses for devload-type gadgets that load device drivers after the system's startup: short of inspecting the SETVER list directly and entering a false version number in the PSP by hand, these utilities would have to misappropriate int 21h function 4B05h (Set Execution State, a new and very poorly drafted function discusssed in Chapter 5), pretending that the device driver just loaded as an overlay is in fact an executable program.

Calling the Kernel

SYSINIT must have explicit knowledge of at least one code address in the DOS kernel—the calling point for initialization. Even here, the possibility that kernel code will be executed in the HMA requires special anticipation. In versions before DOS 5, the MSDOS.SYS file is a simple binary image ready for loading at a paragraph boundary. Initialization is simple: SYSINIT calls the kernel at offset 0000h with respect to the load segment. Addresses in the HMA cannot be normalized like this, however. For example, the byte at the 1MB boundary may be addressed by FFFF:0010 or even F001:FFF0, but not with an offset of zero. As a result the calling point for MSDOS.SYS initialization is addressed with a non-zero offset in DOS 5, although it still lies at the start of the file's binary image. In the particular case of MSDOS.SYS, the addressing base is recorded in the file as the word at offset 03h—immediately after a jump instruction to the initialization code. If, for instance, the MSDOS.SYS file is temporarily at the paragraph boundary with segment address seg and the word at address seg:0003 is 3DD0h, then the calling point is (*seg* minus *03*DD):3DD0 instead of *seg*:0000. Continuing the example, it is the segment *seg* minus 03DD which would be used to form the full address of the routine associated with the kernel code's relocation.

The addressing base for the kernel code is constrained in two directions by several considerations. Most obviously, the addressing base must be large enough to reach past the IO.SYS code from segment FFFFh when DOS uses the HMA; thus,

```
sizeof (IO code) ≤ addressing base
```

On the other hand, the larger the addressing base for the kernel code, the lower the segment that will be used when the code is executed. This segment must not be so low as to make it impossible to address the last byte of kernel code; thus,

```
addressing base + sizeof (maximum MSDOS code) ≤ FFFFh
```

Finally, there is a constraint arising from the possibility that the kernel code may end up in low memory. The addressing base for the kernel code cannot be too large now, lest it require a "negative" segment; thus,

```
addressing base ≤ 0700h + sizeof (minimum IO data) +
sizeof (MSDOS data) + sizeof (IO code)
```

The released DOS versions choose the addressing base to barely meet this last inequality, which has the practical effect that when **dos=low**, the segment used while executing kernel code is very small, the exact value depending on how much IO.SYS data has been retained.

Initializing the Default Device Drivers

As presented in Table 3-1, the information passed to the DOS kernel to describe the results of IO.SYS initialization is deceptively simple. The default CON device driver (whose address is passed in DS:SI) is just the first in a chain of drivers in the low-memory portion of the IO.SYS file. It is understood here that device drivers are "addressed" by their headers. The bulk of the device-driver code is in the relocatable IO.SYS segment. The chain is assumed to be ordered such that the clock device driver (the one with bit 3 set in its attribute word—DOS *never* consults the CLOCK$ device by name) appears before any block device driver.

The DOS kernel initializes each device driver in the chain. To some extent this is little more than a formality, as most of the hardware configuration has already been performed as part of IO.SYS initialization. For the default block device driver, however, (and in principle there may be more than one such driver) the initialization call returns non-trivial information. It is here that the DOS kernel discovers how many units the default block device driver supports and what each of these units looks like physically. Each unit becomes a physical DOS drive with its own letter and its own Drive Parameter Block (DPB)—the kernel's representation of the unit's logical characteristics. Details are taken up in Chapter 16. For present purposes it suffices to know that DOS builds these DPBs at the end of the kernel's data segment.

Interrupt Vectors for DOS Services

In essence interrupt vectors 00h and all those from 20h to 3Fh inclusive are DOS's property. In one way or another, either IO.SYS or MSDOS.SYS claims the corresponding entries in the Interrupt Vector Table. Non-trivial handlers are provided for interrupts 00h, 20h, 21h, 25h, 26h, 27h and 2Fh. With the exceptions given below and of int 29h (for which a handler is installed by IO.SYS as part of the default CON device driver), all the other vectors are pointed at an IRET instruction.

If a **dos=high** regime is configured then all the handlers will require stubs in low memory to shield them from A20 effects. On the other hand, it is desirable (if only for elegance and compatibility with earlier versions) that this stub code not be used if **dos=low**. At this early stage vectors are pointed directly at their corresonding handlers in the kernel code segment. It has been observed already that SYSINIT must give high priority to discerning the **dos** setting; as soon as **dos=low** is discovered, the kernel code segment will be moved to low memory and the stub code will not be hooked into interrupt chains needlessly. If **dos=high** is indicated in CONFIG.SYS then SYSINIT has no choice but to install the low-memory stubs straightaway. It cannot defer this until the kernel code is actually moved to the HMA, since by then user-supplied software might have already hooked the vectors.

Until an XMS server is installed and the **dos=high** regime is known to be achievable, the stubs should act just to divert control from low memory to the appropriate handler in the code segment. Once the XMS server is found and the system code segments are moved to the HMA, these stubs must check the A20, enable it if necessary, and then divert control. This flexibility is arranged by giving the stubs the following structure:

```
DataSegmentInt21h:
                    jmp         short @f
                    call        CheckAndEnableA20
@@:
                    jmp         dword ptr cs:[CodeSegmentInt21h]
```

When the time eventually comes to enable the A20 checking, the two-byte short jump is simply patched over with two NOP instructions.

For historical reasons entries for interrupts 22h, 30h, and 31h are used for purposes unconnected with the formal role of the Interrupt Vector Table. DOS uses the entry for int 22h as storage space for the address to which control must be returned when the current process is terminated. Instead of installing handlers for interrupts 30h and 31h, the location 0000:00C0 is used for a far jump to a CP/M-style entry point for DOS functions 00h to 24h. Every PSP is encoded with a far call to 0000:00C0, or an equivalent logical address, at offset 05h to support the way that CP/M programs obtain operating systems services. Readers who think that CP/M compatibility surely belongs in the waste bin of history and is no longer pursued actively may amuse themselves by wondering why the relocatable IO.SYS segment (which may end up at the start of the HMA) sets aside five bytes for duplicating the CP/M jump at what may become FFFF:00D0.

First Steps to Memory and Process Management

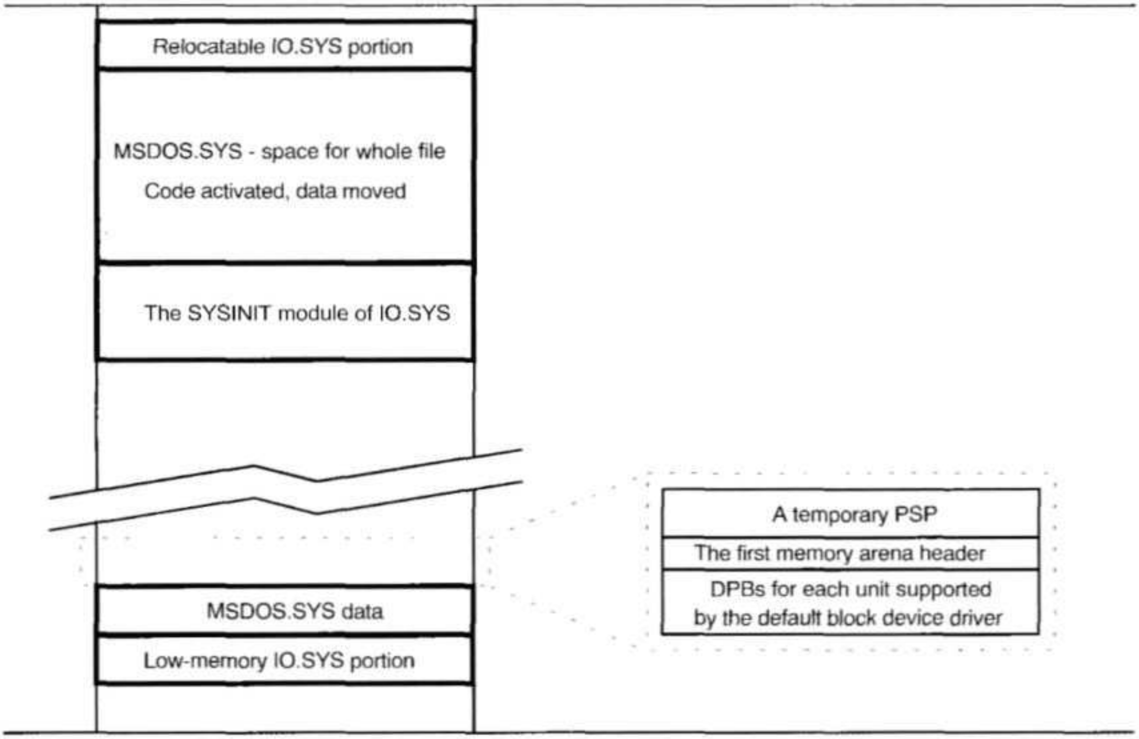
Figure 3-3 depicts the memory situation immediately after MSDOS.SYS initialization. The kernel data is in low memory, followed by the DPBs for each unit supported by the default block device driver. Memory management functions are supported at this stage. Note especially that the chain of memory blocks begins above the system's data areas. The single memory block is regarded as free and extends to the top of conventional memory. The DOS kernel places a barely functional PSP at the start of the memory block.

This PSP has no environment and no parent PSP. Child processes may be started but it is not imagined that this first PSP will ever be terminated. The PSP's handle table is constructed so that the default CON device may be accessed through the stdin, stdout, and stderr handles—all of which are set to index the first of the kernel's five built-in SFTs (already opened with a reference count of three immediately after the device's initialization).

Booting DOS Externally

Immediately upon regaining control from the DOS kernel, SYSINIT resolves a small memory management feature new to DOS 5 and which this discussion has so far omitted.

Figure 3-3: Memory Layout after MSDOS.SYS Initialization


Relocatable IO.SYS portion
MSDOS.SYS - space for whole file
Code activated, data moved
The SYSINIT module of IO.SYS
A temporary PSP
The first memory arena header
DPBs for each unit supported by the default block device driver
MSDOS.SYS data
Low-memory IO.SYS portion

Figures 3-2 and 3-3 show the system modules at the very top of conventional memory, with just a small break representing the single paragraph which may be needed for an arena header if DOS is to take control of upper memory. It is understood that the top of conventional memory is located by inspecting the word at 0040:0013, either directly or through the BIOS int 12h interface. The feature omitted so far is that whenever any part of the DOS 5 version of IO.SYS plans to move material toward the top of memory, it first tests an int 2Fh function (see Table 3-2) to see if some other software has already reserved space there. This applies in the following situations:

- The IO.SYS loader section relocates itself from 0070:0000 to the top of memory so that it can read the rest of the IO.SYS file from disk to 0070:0000. It also needs space in which to hold an FAT sector while following the IO.SYS file's cluster chain.
- The IO.SYS initialization code requires buffers for reading partition sectors while configuring the default block device driver and when loading the MSDOS.SYS file (again, to follow the file's cluster chain).

- The SYSINIT module relocates itself and other system components to the top of conventional memory, where it will also build temporary objects, as discussed shortly.

In all three cases if IO.SYS sees the letters "RPL" three bytes immediately after the address given by the int 2Fh vector, it uses the interface presented in Table 3-2 to see how far it should "lower its sights" before using space near the top of conventional memory. See Figure 3-4.

Table 3-2: Int 2Fh Support for Booting DOS 5+ Externally

INPUT:

AX	4A06h
DX	segment address for ceiling of conventional memory

OUTPUT:

dx	adjusted segment address for ceiling of conventional memory

IO.SYS depends on the interface to preserve registers CL, SI and DI.

It should be stressed that IO.SYS uses this interface only if the int 2Fh vector points to a clearly appropriate handler. If DOS is the only operating system on the computer, then this situation can arise only through an int 19h warm boot. Any stronger startup or restart conditions involve some form of BIOS initialization, which presumably resets the int 2Fh vector. Given the lengthy analysis of int 19h in Chapter 2 and its widespread disregard by DOS's developers, it is hardly credible that int 2Fh function 4A06h is intended to help some piece of software that aims to preserve an area of memory across an int 19h warm boot.

A much more plausible explanation for this interface's existence is that it indicates that DOS is prepared for co-operation with an external operating system. The scheme presently serves to allow diskless workstations to be rebooted while remaining under the control of the network software; RPL stands for "Remote Program Load."

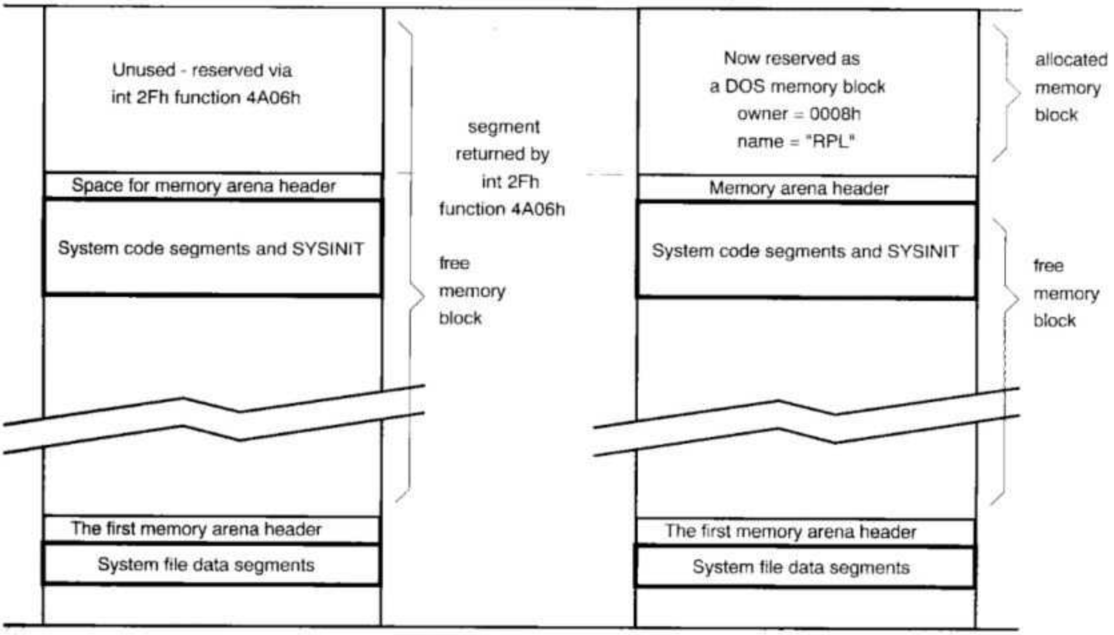
But the RPL need for memory in DOS's RAM area is more general. Whatever the interface may have been designed for particularly, it would also be useful in a system that boots DOS VMs independently of other VMs or native tasks.

The SYSVARS Structure

For SYSINIT to build kernel objects like disk buffers and Current Directory Structures (CDSs), both for temporary purposes now and for the full configuration later, it must know quite a lot about the DOS kernel's inner workings (as opposed to the kernel's large-scale internal organization). Information that SYSINIT gains about the DOS kernel and which is useful not just for initialization is gathered into a structure, the address of which

SYSINIT returns in es:di. This SYSVARS or SysInitVars structure (see Chapter 2) is arguably the most important in low-level DOS programming. It is identical to the so-called "List of Lists" whose address is returned by undocumented int 21h function 52h.

Figure 3-4: Adjustment of Memory for RPL Program Immediately After MSDOS.SYS Initialization


Unused - reserved via int 2Fh function 4A06h
Space for memory arena header
System code segments and SYSINIT
The first memory arena header
System file data segments
segment returned by int 2Fh function 4A06h
free memory block
Now reserved as a DOS memory block owner = 0008h name = "RPL"
Memory arena header
System code segments and SYSINIT
The first memory arena header
System file data segments
allocated memory block
free memory block

Enabling the File System

All the interrupts used for the system interface are in place by the time SYSINIT regains control after MSDOS.SYS initialization. However, the services provided by this interface are not yet stable. Apart from the incomplete nature of process management, there are still some structures that need to be set up before the DOS file system is safe to use. SYSINIT sets about building these underneath its own position in memory, as shown in Figure 3-5. The main requirements for use of the file system arise as follows.

Disregarding remote file systems, which are not yet an issue, DOS cannot perform file I/O without a **buffers** system in which to hold disk sectors. DOS recognizes that since it must have space in which to hold at least one sector (be it an FAT or directory sector loaded into memory while searching through the directory tree or a data sector held in memory while software reads/writes in smaller units than whole sectors), then it assists file access to have the kernel maintain a private cache of the sectors used most recently.

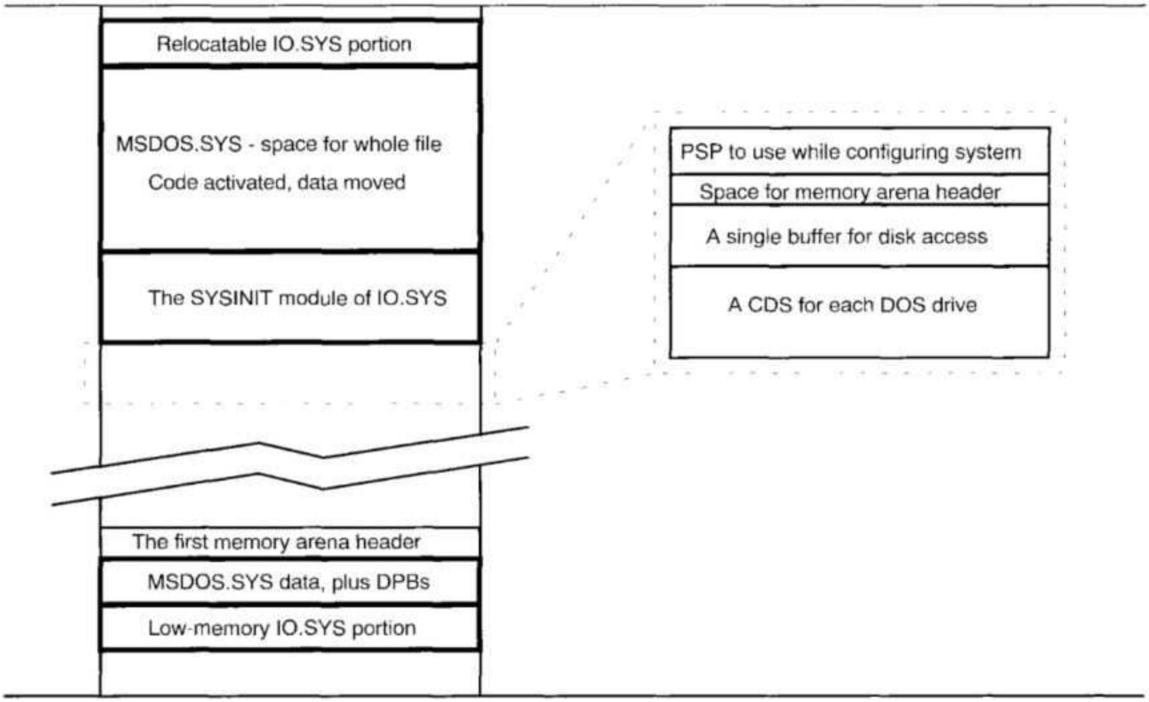
Although a substantial set of sector-sized buffers need not be configured until later, at least one such buffer must be in place before DOS can work with files.

MSDOS.SYS initialization creates DPBs for each unit on the default block device driver. However, the logical correspondence between drives and physical storage devices is held in an array of CDSs. Drive specifications are always dereferenced through the CDSs, but these have yet to be built.

The default IRET instruction for int 24h does not conform to the interface expected for critical error handling. Given the fragile state of the temporary PSP, it is vital that an int 24h exception not be capable of triggering a process termination (and therefore that no int 21h functions capable of generating a critical error be used until a proper handler is installed).

To complete the current picture of memory layout note that it is convenient to move the PSP to the top of memory with the other temporary system objects:

Figure 3-5: Memory Layout After Enabling the File System


Relocatable IO.SYS portion
MSDOS.SYS - space for whole file
Code activated, data moved
The SYSINIT module of IO.SYS
PSP to use while configuring system
Space for memory arena header
A single buffer for disk access
A CDS for each DOS drive
The first memory arena header
MSDOS.SYS data, plus DPBs
Low-memory IO.SYS portion

Observe that the CDSs are placed at the bottom of the temporary objects high in memory. Installing block device drivers increases the number of DOS drives and hence the number of CDSs required. When the system finishes loading device drivers a final set of

CDSs are built at a more suitable location toward the bottom of memory with other system tables. Until then, SYSINIT must either allow a fixed amount of space for a possible 26 CDSs (slightly more than 2K) or the CDSs must be allowed to grow downward.

Processing CONFIG.SYS

Having both arranged the DOS kernel's initialization and created a few additional system structures and work areas, SYSINIT is now ready and able to operate on files. In particular it can examine CONFIG.SYS and begin configuring the system to the user's specifications. During the loading of CONFIG.SYS and, indeed, while loading device drivers, DOS is running as if **buffers=1**. Files are managed through the usual int 21h functions, but access is slow, especially to a file that lies very deep in its drive's directory tree or consists of many discontiguous portions. Both these conditions force frequent consultation of FAT and/or directory sectors.

SYSINIT opens the configuration file under the name "\CONFIG.SYS", having just selected the boot drive as the default. The file's existence is not mandatory. If the file cannot be opened then SYSINIT configures the system using default values. For the most part the system is configured inflexibly, although some attempt is made to adapt the **buffers** and **stacks** systems to the hardware.

A buffer for the file's contents is prepared underneath the temporary disk buffer and the CDSs are redrawn. Not unreasonably, the file size is taken modulo 64K. The whole file is read into memory, but the contents are truncated if an end-of-file character (1Ah) is found.

The Parsing Buffer

All letters 'a' to 'z' throughout the buffer are capitalized before parsing begins. Analysis is conducted one line at a time. The aim is to reduce each line to a single-character token denoting a keyword, followed by some number of characters representing that keyword's value (perhaps with white space stripped out) and then the terminal line-feed. In principle each line's representation is smaller than the original line. The buffer is therefore parsed in place.

The description given here applies to DOS 4 (which saw the introduction of new parsing code) and higher. Some variations will be noted for DOS 6, which sees the introduction of support for multiple configurations that can be selected at boot-time. One such variation must be noted already: So that text for menu options may be displayed in a mixture of upper and lower case, as seems likely to be the user's preference, the DOS 6 version of SYSINIT delays the capitalisation. Indeed, for some keywords, capitalisation is skipped entirely.

Earlier versions may also deviate slightly. The great difference is that they do not support single-line comments. The undocumented **comment** statement defines a character or pair of characters that direct the parser to skip the remainder of the current line. The comment indicator may be changed, but not undefined, mid-way through the parse operation. Uniquely among valid CONFIG.SYS keywords, **comment** acts like a preprocessing directive.

Tab, carriage-return, line-feed, space, and the equals sign cannot be accepted as the first character of a comment indicator. The equals sign is permissable as the second character. In general the separators between parse elements are tab, space, comma, semicolon, and equals sign. In many cases the null byte and forward slash also serve as separators, which situation will be indicated by the term "wide definition."

Though not of vital importance, the parser tends to assume that the CR and LF characters will only occur together and in that order. CONFIG.SYS should not be prepared with the end of a line being represented by just LF or by LF-CR rather than CR-LF.

Tokenization

For each line leading control characters and spaces are skipped. Comments are recognized and cause the whole line to be disregarded. The first non-trivial characters are postulated as a keyword—extending up to but not including the next carriage-return, line-feed, or separator (wide definition). During the examination of keywords, the parser does not inspect characters to see if they introduce comments. If the keyword is valid, then the corresponding token (see Table 3-3) is stored as output. If not then the token 'Z' is stored and the rest of the line is ignored.

A slight variation for DOS 6 is that if the first non-trivial character in a line is a semicolon, the line is tokenized with '0' and a formal keyword is not sought; in other words, such a line is treated as if it had begun with **rem**. A keyword is terminated by a CR, LF or wide-definition separator, as before. An exception is that the character '[' is accepted as a keyword, no matter what characters follow it.

Another feature new to DOS 6 is that any keyword may be terminated by a question mark. In effect, the question mark indicates an alternative form of the base keyword; it is treated as part of the keyword for subsequent parsing and represented in the output buffer by setting the high bit in the corresponding token.

Table 3-3: CONFIG.SYS Tokens

TOKEN	KEYWORD	AVAILABILITY
0	rem	4+
1	switches	4+
A	menudefault	6+
B	buffers	
C	break	

Table 3-3: Continued

TOKEN	KEYWORD	AVAILABILITY
D	device	
E	menuitem	6+
F	files	
H	dos	5+
I	install	4+
J	ifs	4 only
	include	6+
K	stacks	
L	lastdrive	
M	multitrack	4+
N	numlock	6+
O	submenu	6+
P	drivparm	
Q	country	
R	menucolor	6+
S	shell	
U	devicehigh	5+
V	set	6+
W	cpsw	4 only
	installhigh	6+
X	fcbs	
Y	comment	4+
Z		

General Parsing

From here, the parsing rules diverge. The keywords **comment**, **device**, **devicehigh**, **ifs**, **install**, **shell,** and **switches** require special consideration. The general case is to first skip any additional separators (not including the null byte or forward slash) and then copy the remainder of the line as output. In each of these two stages there is some subtlety.

The treatment of additional separators is summarized in the following table, which presents special cases for elimination first, leaving the general case for last:

keyword LF	*token LF*
keyword CR [rest_of_line] LF	*token CR [rest_of_line] LF*
keyword ws [[ns]..] [rest_of_line] LF	*token [rest_of_line] LF*

where *ws* and *ns* represent separators by the wide and narrow definitions, respectively, and *rest_of_line* denotes any string of characters not containing a line-feed. The parser tests for single-line comments while skipping the optional separators.

Comments are also recognized while copying the rest of the line. If found, they and all remaining characters up to but not including the terminating line-feed are disregarded. There is an exception because the parser supports a quoting scheme. Between double-quotes in this copy-rest-of-line stage, single-line comments are suspended. It is perhaps not obvious that this quoting system extends across lines. For instance,

```
comment //
rem This is an isolated double-quote "
device c:\dos\ramdrive.sys 1024 /e     // drive F:
rem This is its belated partner "
```

means that the potentially helpful annotation about the RAM drive's identity becomes an (invalid) argument to the device driver instead of being disregarded by the parser.

Devicehigh

The rules for the copy-rest-of-line stage are modified very slightly for the **devicehigh** statement. Each CR encountered before the terminating LF generates both a space and CR as output. This extra space is also generated for the LF character unless exactly one CR has been copied. The most common effect of all this is that a CR-LF combination to end a **devicehigh** line becomes a SPACE-CR-LF in the parsed buffer.

Comment

Of course the **comment** directive must be parsed differently from the others, particularly if it is to support the use of separators as comment indicators, say by some construction such as

```
comment = ;
```

for single-line comments in MASM style. In the following table *ws* represents a wide-definition separator, *tse* stands for any of the comment separators tab, space, or equals, and *tsr* for any of tab, space, or carriage-return. The parsed output is always the 'Y' token followed by the terminating line-feed, so the table shows the pre-processing action taken by the parser.

COMMENT *LF*	skip next line
COMMENT *CR [rest_of_line] LF*	
COMMENT *ws [[tse]..] LF*	
COMMENT *ws [[tse]..]* CR *[rest_of_line] LF*	
COMMENT *ws [[tse]..] c1 [tsr [rest_of_line]] LF*	adopt *c1* as comment indicator
COMMENT *ws [[tse]..] c1 c2 [rest_of_line] LF*	adopt *c1 c2* as comment indicator

Ending a line with just the line-feed character induces the parser to ignore the whole of the next line.

Switches
The first character of the **switches** value is exempted from quoting. Thus, after skipping any separators that follow the keyword, the parser stores the first character as output and only begins to apply the copy-rest-of-line rules from the next character onward.

SWITCHES *LF [next_line LF]*	1 *LF [next_line LF]*
SWITCHES *CR [rest_of_line] LF*	1 *CR [rest_of_line] LF*
SWITCHES *ws [[ns]..] LF [next_line LF]*	1 *LF [next_line LF]*
SWITCHES *ws [[ns]..] char [rest_of_line] LF*	1 *char [rest_of_line] LF*

Again, the abrupt termination of the line (by a line-feed immediately after the **switches** statement) induces unexpected behavior. The whole of the next line is copied as output according to the usual rules about comments and quotes. It is treated as the rest of this line rather than being parsed as a line in its own right.

Driver and Program Commands
With the exception of **devicehigh**, the CONFIG.SYS statements that direct SYSINIT to load a file (**device, ifs** in DOS 4, **install,** and **shell**) are treated specially. Primarily so that the file's pathname will be ready (as a null-terminated string) to pass to the appropriate DOS functions. Although the **devicehigh** keyword introduced with DOS 5 acts in some senses like a variant of the familiar **device**, it does not fit easily into this group because the first element of its value need not be a pathname, it could be the **size** parameter.

The first divergence from the standard parsing is that *all* separators after the keyword are skipped—including nulls and forward slashes. As usual, comments are recognized after the first separator. The first character of the value is stored as output. Even if it is a bare

LF because the parser will not recognize a line-feed in this position as terminal is then followed by the other characters of the presumed pathname—up to but not including the first separator or control character.

The general rule now is that no matter how the pathname is terminated in the text file, its parsed equivalent must be terminated by a null byte.

INPUT FROM TEXT FILE IMMEDIATELY AFTER PATHNAME	**PARSED OUTPUT**
LF	*null LF*
/ [rest_of_line] LF	*null / [rest_of_line] LF*
cs [rest_of_line] LF	*null [rest_of_line] LF*

cs denotes any control character or separator other than the forward slash. The forward slash receives special treatment to support those whose habit is to follow commands immediately by their switches. In the general case the first separator is replaced by a null byte. Additional separators, if present, are copied as part of the rest of the line—in contrast to optional separators between the keyword and pathname, which are discarded. Incidentally, when SYSINIT initializes device drivers, it converts the null byte to a space.

The [keyword in DOS 6 and higher is parsed in the default manner: The rest of the line is copied as output. The only unusual quality is that characters are not capitalized. Only narrow separators are skipped after the [keyword.

[*LF*	[*LF*
[*[[ns]..] [rest_of_line] LF*	[*[rest_of_line] LF*

For all but the **include**, **menudefault**, **menuitem,** and **submenu** commands, lower case letters on the rest of the line are capitalized, except that, for **set**, capitalization stops on encountering an equals sign.

In DOS 6, if the keyword is terminated by a forward slash, then examination of the rest of the line (in order to find the keyword's value) begins with the slash. This is not true in earlier versions, where such a slash would have played its role as a separator and thus not enter into further consideration. The most plausible use for this feature is that a line such as **switches /k** is parsed the same way, with or without a space (or other wide separator) between the keyword and its value.

keyword LF	*token LF*
keyword CR	*token CR*
keyword /	*token /*
keyword ws	*token*

Of the new keywords, **installhigh** is treated with **install**.

Configuration Block Analysis

In DOS 6 and higher, the CONFIG.SYS file may be subdivided into configuration blocks, each of which begins with a heading and extends up to but not including the next heading (or trivially, to the end of the file). A heading is a line introduced with the [keyword. It contains a title for the block and consists of all characters from the first non-separator (narrow definition) up to but not including the next CR, LF, separator (wide definition), or square bracket (left or right). Title comparisons are always independent of case.

Global Properties

Three commands may affect the analysis of configuration blocks. Most obviously, these include **switches /f** and **switches /n** (possibly in one statement, along with other switches). In addition, SYSINIT processes **numlock** commands occurring before the first block heading.

Consulting the Keyboard

Unless **switches /n** is found, SYSINIT checks the keyboard (through BIOS functions) for direction. This includes the case where **switches /n** is not found for the trivial reason that no CONFIG.SYS file exists.

Unless **switches /f** was found, SYSINIT waits two seconds before checking the keyboard. The following cases are recognized:

- F5 is found waiting in the keyboard buffer;
- F8 is found waiting in the keyboard buffer;
- Either shift key is held down.

In the first two cases, the keyboard token is removed from the buffer. Given F5 or a shift key, SYSINIT prepares a "/D " switch for the **shell** and skips the processing of menu blocks.

Menu Blocks

A menu block is a special type of configuration block containing commands that refer to other configuration blocks (including other menu blocks) or indicate how SYSINIT should present these references as a menu on the screen.

The first configuration block to be inspected has the specific heading **[menu]**. No configuration blocks are inspected unless there exists one with "menu" as its title.

The only non-trivial lines a menu block should contain are those introduced with the **menucolor**, **menudefault**, **menuitem**, **numlock**, **rem,** or **submenu** keywords. The presence of any other keyword causes SYSINIT to declare an error and wait for a keystroke to indicate acknowledgement before it resumes processing the block.

Menu Block Commands

The **menucolor** keyword takes one or two numerical arguments: a foreground and a background color, respectively, using the usual scheme of video attributes. If two arguments are given, they must have exactly one separator (wide definition) between them. The second argument is optional: It defaults to a black background.

The **menudefault** keyword also takes one or two arguments: a string identifying the selection and an optional, numerical timeout. There is more flexibility over the separation of arguments here. The string argument stops at the first separator (wide definition), CR, LF, or square bracket, and there may be any number of wide separators before the numerical argument, which is understood as a number of seconds and coerced to a maximum of 90. The timeout argument is ignored if the running timeout counter is already non-zero.

The **numlock** keyword is processed on discovery. The command is ignored if its argument is invalid, but only the first two characters of the on|off value are checked.

SYSINIT ignores **menuitem** and **submenu** after recording details for nine of them. For both commands, the first argument must be a string that exists elsewhere in the file as a configuration block title: If not, SYSINIT raises an error requiring a keystroke before resumption. The two arguments are subject to the same separation criteria as **menudefault**. The only difference between the two commands is that **submenu** should refer to another menu block and that **menuitem** should refer to a configuration block containing system commands (but note that the distinction becomes meaningful only when the corresponding reference is selected).

Option Selection

After working its way through a whole menu block, SYSINIT examines the **menuitem** and **submenu** statements. If the block contains no references to other configuration blocks, the recursion through the heirarchy of menu blocks is terminated.

Otherwise, SYSINIT prepares to offer the references to the user as a screen menu. A default reference may have been nominated through a **menudefault** statement; if so, SYSINIT searches the CONFIG.SYS buffer for a configuration block title to match that statement's first argument. If the menu block contains no **menudefault** statement or if no match exists, then SYSINIT adopts the first reference as the default.

This default item or submenu is selected automatically if a timeout of zero has been declared through a **menudefault** statement in the block and the F8 flag is off.

Otherwise, SYSINIT resets the current video mode, selects video page zero, and fills the screen with the chosen color. Incidentally, the menu color has a global quality: Unless it is changed explicitly, it is retained as the recursion proceeds and even if the recursion is restarted.

In the absence of **switches /n**, the line

```
F5=Bypass startup files   F8=Confirm each CONFIG.SYS line [ ]
```

is added to the bottom of the menu screen.

For each **menuitem** and **submenu** line (up to a maximum of nine) in the menu block, the corresponding text (second argument if given, else the first) is numbered and displayed. The text stops with the first control character or dollar sign or after displaying a character at the 0-based column number 78. A tab is not counted as a control character for this purpose, but is instead converted to a single space. The default item is given a white background of the same intensity (or more likely, the same blink state) as other items; if this would result in an unreadable white-on-white, the foreground intensity is changed to ensure visibility.

The number corresponding to the default item is also shown at the "Enter a choice: " prompt.

In the absence of **switches /n**, the space between square brackets in the F5/F8 prompt is filled with a Y or an N, depending on the current state of F8 confirmation. If **switches /n** was present, this line of the menu screen is blanked out.

Unlike the global case, keys are sought at stdin, not through the BIOS.

If the timeout scheme is operating (that is, a non-zero timeout was specified in a **menudefault** statement), the SYSINIT displays a "Time remaining: " notice after the prompt. SYSINIT tracks the time by consulting the DOS clock each time it discovers that no character is waiting at stdin. Curiously, only seconds are monitored and the algorithm may behave strangely if keyboard checking is delayed (say, by pressing the Pause key in the hope of deferring a decision).

Keys may be thought of as acting in two groups. Keys in the first group change the selection, pending the use of a key in the second group to show what should be done with the selected information:

KEY	EFFECT
Number	change default to the numbered option
Up	change default to the next lower-numbered option
Down	change default to the next higher-numbered option
F8	toggle the confirmation flag

KEY	EFFECT
CR	select default option and confirmation flag
Backspace	return to top-level menu
F5	skip configuration file processing

F5 and F8 are recognized only if **switches /n** was not given. Pressing F5 has the same consequence here as in the global stage before any menu blocks were inspected: SYSINIT proceeds as if CONFIG.SYS were empty.

If a **submenu** is selected, then the configuration block (already known to exist) with the matching title is found and processed as a menu block. This recursion through a heirarchy of menu blocks ends normally with the selection of a **menuitem** or the selection of a menu block containing no references to other configuration blocks. Abnormal termination is achieved through the use of F5. In effect, the Backspace key restarts the recursion. There seem to be no safeguards against circular references.

The CONFIG.SYS Extract

Having either selected a **menuitem** or reached an essentially empty **submenu**, SYSINIT extracts the CONFIG.SYS lines that comprise the working configuration; that is, all lines before the first block heading, then all lines in configuration blocks that have either the specific heading [**common**] or the heading given as the selected **menuitem**. These lines are transferred to the second parse buffer.

Block Inclusion

During this extraction process, SYSINIT resolves any **include** statements that it encounters in the working configuration. The argument to **include** must be the title of a configuration block but not "menu" or "common." Inclusion is recursive; curiously, there seem to be no precautions against self-inclusion.

It is very important, especially from a compatibility perspective, to understand that if CONFIG.SYS contains any block inclusions or block headings (valid or not), then lines are not simply copied to the final parse buffer. Instead, the command token is followed by the 1-based line number (as a word) before the rest of the line is copied as usual. This allows SYSINIT to discard the first parse buffer but still know the original line numbers.

Multiple Passes

CONFIG.SYS statements are not processed in their order of appearance. Instead, SYSINIT makes several passes over the parsed buffer, extracting only what it needs each time.

Pass One

dos

Act now if **dos=low**, then the system can be arranged compatibly with earlier versions and with all system material as low as possible (underneath the memory arena).

If **dos=high** the DOS kernel must still be told. There are some irreversible changes, so that a **dos=high** configuration that can't make it (for example because an XMS server does not get installed) does not end up looking quite like **dos=low**.

Pass Two

Action taken in order of appearance:

break

country

device

devicehigh

drivparm

multitrack

switches

Note especially that the UMB chain is unlinked after the action stage of this pass.

Values recorded in order of appearance:

buffers

fcbs

files

lastdrive

shell

stacks

Pass Three

Now unused but was introduced in DOS 4 to support installable file systems.

After pass three, SYSINIT acts on the values recorded earlier.

All file handles are closed except stdout. Then CON is opened and duplicated twice. After this, AUX and PRN are opened. The remnant of this is seen in the way that the first SFT usually refers to AUX—but it is not necessarily so.

Finally, some memory block rearrangement: this includes preparation for **install**.

Pass Four

install

Stabilizing the System

Here is the final sequence of events in the system startup:

Final attempt to put DOS in the HMA if **dos=high**.

Duplicate five bytes from 0000:00C0 at FFFF:00D0 for CP/M.

Tidy up memory used by SYSINIT.

Put a VDISK header at 1MB to protect the HMA if **dos=high** (though this is surely unnecessary if int 15h function 88h is by now reporting no free extended memory).

Run the **shell**.

Loading DoubleSpace Early—How Is It Done?

Immediately after the few steps with which it completes the DOS kernel's preparation of the file system, the DOS 6 version of SYSINIT loads and initializes DBLSPACE.BIN, the driver for transparent disk compression. The most obvious effect is that DBLSPACE.BIN is sought ahead of the CONFIG.SYS text file. Because of this early initialization and subsequent manipulations while waiting for upper memory to become available, or for all block devices to have been assigned their expected drive letters, only the system files, DBLSPACE.BIN itself, and a DoubleSpace configuration file need be accessible without decompression. This reduction in the number of files that must be exempted from compression under DoubleSpace in comparison with pre-existing schemes (such as Stacker) designed and marketed by independent vendors is the main user-level benefit from Microsoft's incorporation of disk compression into DOS 6.

In a truly integrated disk compression scheme (such as one might expect from descriptions of DBLSPACE.BIN as a new system file), DBLSPACE.BIN would be some sort of optional extension to IO.SYS, intended to give the default block device driver the means to use disks with a new format that indicates compressed data. This would have several advantages by way of greater transparency, most notably that recognition could be automatic. The system's ability to mount compressed drives at startup would not depend so crucially on the drive assignments set down in the DBLSPACE.INI file, as they do in the scheme Microsoft has actually implemented.

The plan is to initialize DBLSPACE.BIN very much like a self-standing block device driver, but without having to wait for a **device** statement in the CONFIG.SYS file.

Remember that ordinarily, SYSINIT reads CONFIG.SYS and looks for a **dos** statement to determine what steps must be either taken immediately or deferred. Device drivers are loaded in the order that **device** or **devicehigh** statements are found during a second pass, through a parsed representation of the CONFIG.SYS file. Before starting this second pass, SYSINIT obtains the largest available block of memory. All device drivers end up in this one block unless they are loaded into upper memory instead. At the end of the second pass, SYSINIT builds various system structures in low memory; these also go into that one block.

The new treatment has the appearance of a hack. Immediately before the instructions where the DOS 5 implementation of SYSINIT would have read the CONFIG.SYS file, the DOS 6 version of SYSINIT obtains the block of memory just as it would normally, but earlier. If DBLSPACE.BIN is not found (it must be in the root directory of the boot drive) or does not install, then the block is released as if nothing had happened.

On the other hand, if DBLSPACE is retained, SYSINIT trims the memory block (according to DBLSPACE's directions, as will be seen shortly) and starts with a new block later. The memory block containing whatever DBLSPACE.BIN has left behind in low memory is assigned as a DOS System Data block, i.e., its owner field is set to 0008h, and the characters 'SD' are set as the owner's name. To a very large extent, this assignment is then made inconsequential by revising the variable in which DOS keeps the address of the first memory block, so that the memory arena chain starts after the block containing DBLSPACE. In practice then, the memory block allocated for the DBLSPACE.BIN stub in low memory is not visible to applications and does not appear in memory maps.

Modified Device Driver Initialization

To distinguish this early activation from normal device driver initialization, SYSINIT calls DBLSPACE through a specially modified interface. To be recognized for this purpose, DBLSPACE.BIN file must have the signature word 2E2Ch at offset 12h in its image, that is, immediately after the device driver header. If this is found, then SYSINIT calls DBLSPACE at offset 14h, where (presumably) a jump instruction diverts control to more substantial (and discardable) initialization code.

Table 3-4: Far Call from SYSINIT to Initialize DBLSPACE.BIN

INPUT:

AX	joint IO.SYS and DBLSPACE.BIN version number; major version in low byte, e.g., 0006h
ES:BX	address of device driver request packet prepared for initialization (only non-zero fields listed below)

offset	size	description
00h	byte	length of request packet: 18h bytes
0Eh	dword	address of first byte beyond area of memory available to driver
12h	dword	address of null-terminated pathname for driver, specifically "\DBLSPACE.BIN"
16h	byte	0-based number of first available DOS drive

SUCCESS:

cf	clear
ax	0000h

device driver request packet contains data from driver

offset	size	description
0Dh	byte	number of units supported by driver
0Eh	dword	address of first byte beyond area of memory retained by driver
12h	dword	address of array of near pointers to BPBs for supported units

Table 3-4: Continued

FAILURE (WRONG VERSION):

cf	set

FAILURE (OTHER REASON):

cf	clear
ax	non-zero

No assumptions are made about register preservation. Put another way, as far as SYSINIT is concerned, DBLSPACE.BIN may corrupt all registers.

SYSINIT announces a "Wrong DBLSPACE.BIN version" if this initialization returns with the carry flag set. Returning non-zero ax (with the carry flag clear) is also failure, but no error message is generated.

A Cleanup Problem in SYSINIT

The current implementation of SYSINIT does not deal correctly with a situation in which DBLSPACE.BIN declares a successful initialization through the above interface but fills the request packet with values that SYSINIT either cannot accept or interprets as a direction to discard the driver.

Note in particular that if DBLSPACE cannot install, then it must return one of the failure conditions described in Table 3-5 below; it should not just prepare a request packet in the usual manner of a block device driver reporting its failure to initialize (no supported units or a break address equal to the driver's load address). Of course, this situation can be ruled out from the interaction between SYSINIT and DBLSPACE.BIN since the interface is a private arrangement between the two, and DBLSPACE presumably knows to declare its failure correctly.

Consider however, that the early initialization of DBLSPACE must seem attractive in some quarters. The appearance of "fake" DBLSPACE.BIN programs or worse, of software that replaces all DBLSPACE.BIN strings in IO.SYS with the name of some other program, is probably not far off. It can only be hoped that such DBLSPACE copycats are aware of all subtleties. Unfortunately, the history of competition within the PC industry is that imitative software too often shows that much less attention is paid to software research than to market research, advertising and if need be, to law suits or to grumbling before regulatory bodies.

Room for trouble exists in the way that DBLSPACE, like any device driver, has only a restricted picture of the system into which it is being added. It is conceivable that DBLSPACE may be satisfied with an initialization that SYSINIT (in possession of the overall picture) rejects. This, too, is hardly likely to occur without some oversight on DBLSPACE's part. After all, the factors involved in SYSINIT's decision whether or not to retain an initialized device driver are few and device drivers are in a position to check them before declaring a successful initialization. It may be reasonable then, to suppose that SYSINIT need not guard itself against such things as DBLSPACE trying to support more units than can be accommodated within DOS's limit of 26 drives. Still, the checks are in place; SYSINIT does have the means to reject what DBLSPACE has thought to be a successful initialization and the consequent clean up is performed incorrectly.

What happens is that SYSINIT releases the memory into which it loaded DBLSPACE but continues to think that DBLSPACE is still available at the calling point for its special interface. After reading CONFIG.SYS, SYSINIT will direct DBLSPACE to hook interrupt vectors even though SYSINIT will soon start loading device drivers into the memory that DBLSPACE had occupied. The result, if DBLSPACE somehow manages to return unacceptable request header fields despite thinking its initialization to have been successful, is almost certainly a system crash.

This incomplete cleanup is, perhaps, nothing more than a purist's technicality. Consider however, that even if there is no way that the current implementation of DBLSPACE.BIN can trigger the problem, the problem's mere existence serves as a first indication that even the vaunted integration of DoubleSpace with the system was at best a sloppy endeavor.

Moving DBLSPACE.BIN Away

When SYSINIT regains control from a successful DBLSPACE initialization, its first priority is to move DBLSPACE.BIN to some location high in conventional memory (along with various bits and pieces of the system files) in order to make room for other software that must be packed in low memory. The main hope is ultimately to move the fairly large DBLSPACE.BIN image to upper memory, but since such memory is not available to DOS until an appropriate memory manager has been loaded, the final destination for DBLSPACE will not be known for some time.

SYSINIT should not need to know anything of DBLSPACE's internal structure: only that DBLSPACE will require some space near the top of memory for a while and that the driver may want to retain some stub at the load address. This separation is embodied in SYSINIT's next two calls to the special DBLSPACE entry point at offset 14h from the driver's load address.

Table 3-5: Far Calls to Move DBLSPACE.BIN to High Memory

DBLSPACE Far Call 04h - Get Size of Relocatable Portion

INPUT:

BX	0004h

Table 3-5: Continued

OUTPUT:

ax	number of paragraphs SYSINIT should move to top of memory

DBLSPACE Far Call 06h - Relocate

INPUT:

BX	0006h
ES	destination segment

OUTPUT:

ax	address of first segment surrendered in low memory

SYSINIT makes no assumptions about register preservation for either of the above functions.

Thus, SYSINIT determines how much space DBLSPACE will require at its temporary position high in memory, calculates some suitable segment address and then calls DBLSPACE to relocate itself or whatever portion of itself it chooses to move. The one subtlety is that DBLSPACE is asked to move only if the new, higher position will not overlap the original one.

Setting Up Drives for DoubleSpace

As usual for block device driver initialization, SYSINIT checks that the number of units to be supported is not zero, then saves the number at offset 0Ah in the device driver's own header. All being well, each of these units will become a physical DOS drive. SYSINIT checks that the total number of physical drives will not exceed 26. Finally, it inspects the driver's break address.

Table 3-6: DBLSPACE Far Call 02h - Update Initialization

INPUT:

BX	0002h
AH	55h
AL	drives added since last call
CX	segment address of available memory
DX	number of paragraphs of available memory

If DBLSPACE.BIN installs, DOS's memory arena chain is revised so that it begins above the stub left in low memory and the DPBs.

Figure 3-6: Before and After DBLSPACE.BIN Relocation

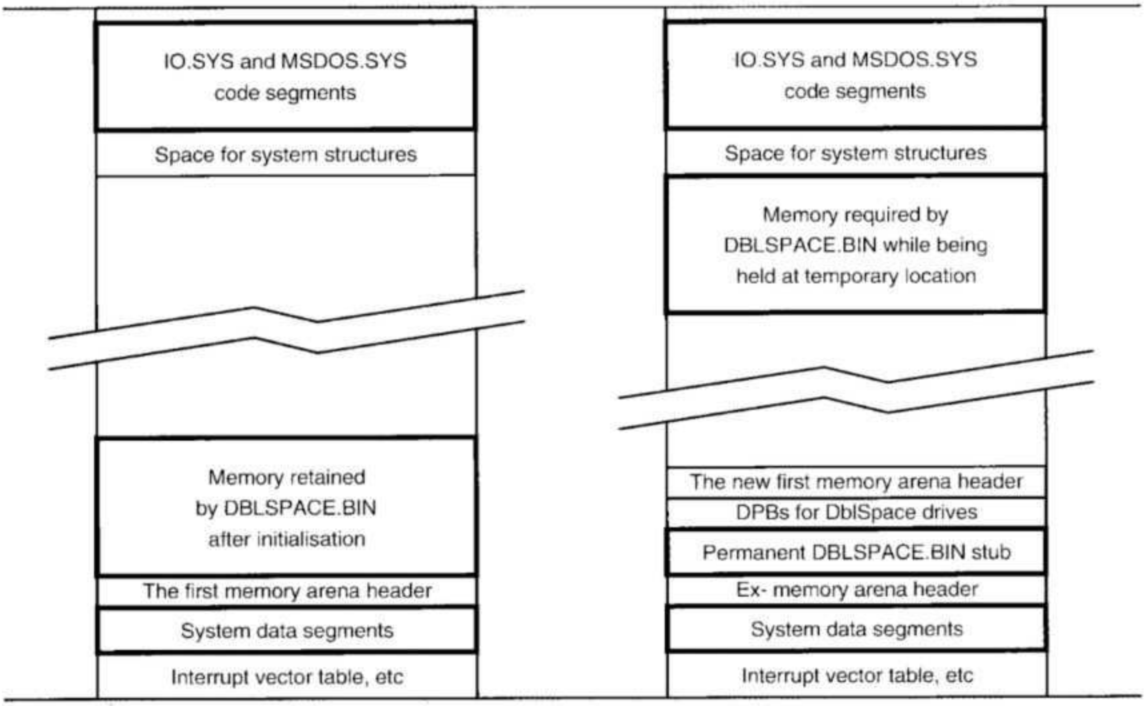

IO.SYS and MSDOS.SYS code segments
Space for system structures
Memory retained by DBLSPACE.BIN after initialisation
The first memory arena header
System data segments
Interrupt vector table, etc
IO.SYS and MSDOS.SYS code segments
Space for system structures
Memory required by DBLSPACE.BIN while being held at temporary location
The new first memory arena header
DPBs for DblSpace drives
Permanent DBLSPACE.BIN stub
Ex- memory arena header
System data segments
Interrupt vector table, etc

Table 3-7: DBLSPACE Far Call 00h - Hook Interrupt Vectors

INPUT:

BX	0000h

Table 3-8: DBLSPACE Int 2Fh API Functions for Final Positioning

DBLSPACE Function FFFFh: Get Final Size Requirement

INPUT:

AX	4A11h
BX	FFFFh

Table 3-8: Continued

OUTPUT:

ax	number of paragraphs required

DBLSPACE Function FFFEh: Move Driver to Final Position

INPUT:

AX	4A11h
BX	FFFEh
ES	segment for final position

CHAPTER 4

Memory Allocation

DOS provides clients with an essentially unlimited number of possible memory allocations. Instead of maintaining an exhaustible set of memory control structures in its own data space, the kernel builds a chain of headers progressing through the memory arena on which applications get to strut their stuff. The memory arena header is one of the best-known structures in DOS programming, despite having an "unofficial" status until its recent documentation for DOS 5.

Allocation Strategy and Upper Memory Linkage

DOS programs allocate from upper memory using the normal DOS memory-allocation service (int 21h function 48h; see Table 4-1). However, to allocate from upper memory requires first setting the memory-allocation strategy with int 21h function 58h.

Note the two-step nature of setting a strategy that either forces or prefers allocation from upper memory. The "loadhigh" and "highonly" allocation strategies are wholly ineffective unless the upper memory link is enabled (see Table 4-1 for DOS functions 5802h and 5803h).

Table 4-1: DOS Function 5800h—Get Memory Allocation Strategy

INPUT:

AX	5800h

OUTPUT:

cf	clear	
ax	memory allocation strategy	
	style of fit masked by 003Fh:	
	0000h	first fit—accept first block that meets request
	0001h	best or tightest fit—find smallest block that meets request
	0002h	last fit—take allocation from the end of the last block that meets request

Table 4-1: Continued

other bit flags control chain selection:

	if upper memory link enabled	if upper memory link disabled
0000h	low first then upper	low only
0040h	upper only	low only
0080h	upper first then low	low only
00C0h	upper only	low only

The memory allocation strategy is maintained as a byte in the DOS kernel's data area. No interface returns the address of this variable, so DOS functions 5800h and 5801h provide the only means of access. The variable lies outside the Swappable Data Area (SDA) described by DOS function 5D06h and is not instanced by Windows. That this and other key DOS variables lie outside the SDA is a crucial point missed by those writers who assert that reliable TSRs can be written by swapping the SDA.

DOS versions lower than 5.0 do not offer an arena chain in upper memory and do not recognize the bit flags for chain selection. Function 5801h accepts arbitrary values in AL, so with improper use of that function, this one may return undefined values in register AL. In earlier DOS versions the memory allocation routines interpret all strategies ≥ 02h as last-fit.

Table 4-1: Continued

DOS function 5801h—Set Memory Allocation Strategy

INPUT:

AX	5801h
BX	memory allocation strategy

SUCCESS:

cf	clear

FAILURE:

cf	set
ax	0001h if BX is not an allowed strategy

DOS versions lower than 5.0 permit arbitrary BX when setting the strategy. The memory allocation routines interpret strategies ≥ 0002h as last-fit.

Table 4-1: Continued

DOS function 5802h—Get Upper Memory Link

INPUT:

AX	5802h	

OUTPUT:

cf	clear	
al	00h	if upper memory link is disabled
	01h	if upper memory link is enabled

The upper memory link is bit 0 of the byte at offset 0089h in the DOS kernel's data segment. As with the memory-allocation strategy byte used by functions 5800h and 5801h, this places it outside the SDA, but unlike the memory strategy, the byte is instanced by Windows.

The boundary segment between the two memory arena chains (UMBHEAD) is recorded at offset 008Ch in the DOS kernel's data segment. Again, this places it outside the Swappable Data Area, and again the word is instanced by Windows. In Chapter 2, Table 2-15 (DOSMGR broadcast 0), the address of UMBHEAD is at offset 0Ch in the instance table returned by DOS.

Table 4-1: Continued

DOS function 5803h—Set Upper Memory Link

INPUT:

AX	5803h	
BX	0000h	to disable upper memory link
	0001h	to enable upper memory link

SUCCESS:

cf	clear	

FAILURE:

cf	set	
ax	0001h	if BX is not an allowed value; or if no upper memory chain exists; or if the chain in conventional memory is corrupt

The function attempts to return ax = 0007h if the memory arena chain is corrupt, but error code mapping (see Chapter 17) is set up so that DOS function 58h can only ever return 0001h in ax when an error occurs.

Table 4-1: Continued

DOS function 48h—Allocate Memory

INPUT:

AH	48h
BX	number of paragraphs sought

SUCCESS:

cf	clear
ax	segment address of allocated memory block

FAILURE:

cf	set	
ax	0007h	if memory arena chain is corrupt
	0008h	if request is greater than size of any free block
bx	if error code is 0008h—size in paragraphs of largest block available	

Among the function's side effects are validation of the memory arena chain and coalescence of adjacent free blocks.

Standard practice for obtaining the size of the largest free memory block is to request FFFFh paragraphs of memory. Very often, and within the system files themselves, it is assumed that this request will fail for lack of a free 1MB block of memory and that a second call will allocate the largest available block. This can go wrong in two ways. Most obviously, the first call may fail for an unanticipated reason (the memory arena chain may be corrupt), but this is likely to be caught on the second call. The second possibility affects code running under the supervision of a DOS extender: The request for 1MB may actually succeed!

Table 4-1: Continued

DOS function 49h—Free Allocated Memory

INPUT:

AH	49h

Table 4-1: Continued

ES	segment address of block to release

SUCCESS:

cf	clear
ax	corrupt

FAILURE:

cf	set	
ax	0009h	if address given in ES does not seem to describe a memory block

When the function checks that ES actually addresses a memory block, it does not follow the chain of memory arena headers from its beginning. It establishes only that the paragraph below ES contains a memory arena header with a valid signature (and then clears the owner field).

When **dos=high** this function may be called once by a newly-started program without causing DOS to enable the A20 on completion of the function.

Table 4-1: Continued

DOS function 4Ah—Set Memory Block Size

INPUT:

AH	4Ah
BX	number of paragraphs sought
ES	segment address of block to resize

SUCCESS:

cf	clear
ax	segment address of block

FAILURE:

cf	set	
ax	0007h	if address given in ES does not seem to describe a memory block; or if memory arena chain between ES and the next allocated block is corrupt
	0008h	if too little free memory is available above ES to accommodate the request
bx	if error code is 0008h—size in paragraphs to which block could have been expanded	

When the function checks that ES actually addresses a memory block it does not follow the chain of memory arena headers from its beginning. It establishes only that the paragraph below ES contains a memory arena header with a valid signature.

Error code 0009h—which might be expected if ES does not address a DOS memory block (and the programmer is relying on the official reference)—can never be returned in ax by this function.

The block at ES is coalesced with all free blocks that lie immediately above it—before the function compares the requested size against the space available. No restoration is attempted if the request is still too large. Therefore, the value returned in bx is the size to which the block *has* been expanded.

The block is reassigned to the current process—an unnecessary action which may even be unexpected, though admittedly it is hard to construct a reason for one process to be resizing another's memory allocations.

A Practical Example—Selecting Memory Blocks by Address

EATBLOCK is a program that obtains a block of memory of a specified size within a specified address range, either permanently or within a shell for testing:

```
EATBLOCK size [base [ceiling]] [/p]
```

All numeric values are interpreted as hexadecimal; size is in paragraphs; base and ceiling are segment addresses. Use size=FFFF to report on the largest free block within the given range.

Ordinarily, EATBLOCK starts a command processor to support testing under the new memory environment. The memory block is released when the command processor exits. The /p switch instructs EATBLOCK to make its allocation permanent and to exit immediately (i.e., without starting a shell).

The code for EATBLOCK is organized roughly as follows:

```
main (MEMMAN\EATBLOCK\EATBLOCK.C)
    ReadHexWord (LIB\READHEX.C)
    _dos_special_allocmem (MEMMAN\EATBLOCK\MALLOC.C)
        _dos_getumblink (LIB\MEMSTRAT.C -> 21/5802)
        _dos_getbreakflag (LIB\BREAK.C -> 21/3300)
        _dos_setbreakflag (LIB\BREAK.C -> 21/3301)
        _dos_setumblink (LIB\MEMSTRAT.C -> 21/5803)
        MaskMemoryRegion (MEMMAN\MEMBLOCK.C)
            AddLargestBlock (memblock.c)
                _dos_allocmem (DOS.H -> 21/48)
                AddBlock (memblock.c)
                    malloc
            PunchHole (memblock.c)
            _dos_setblock (DOS.H -> 21/4A)
            AddLargestBlock (see above)
            _dos_freemem (DOS.H -> 21/49)
            RemoveBlock (memblock.c)
                free
        ReleaseMask (memblock.c)
            _dos_freemem
            free
    _dos_allocmem
    ReleaseMask
AssignBlock (MEMMAN\EATBLOCK\EATBLOCK.C)
RunTestShell (MEMMAN\EATBLOCK\EATBLOCK.C)
    getenv
    PutError
    printf
    spawnl
    strerror
```

EATBLOCK requires the following C source code files:

- EATBLOCK.C is the top-level module containing command-line processing, and so on.
- MEMBLOCK.C is the memory block management engine for this program and for the SPOTLOAD program in Chapter 5.
- READHEX.C is a small module that will be reused. Obviously, it is like the C `atoi` function, but for hexadecimal expressions.
- BREAK.C is another small module that will be reused time and time again.
- MALLOC.C provides the function `_dos_special_allocmem`, which generalizes the C language front-end to int 21h function 48h so that it also takes base and ceiling arguments, giving an address range from which memory is to be allocated.

```
/* MALLOC.C */

/*  This file provides a generalization of DOS memory allocation to allow
    selection of a memory block by location as well as size. */

#include    <dos.h>
#include    <stdlib.h>

#include    <standard.h>

extern WORD MaskMemoryRegion (_segment, _segment, WORD *);  // memblock.c
extern WORD ReleaseMask (WORD *);                            // memblock.c
extern WORD _dos_getumblink (BOOL *);                        // memstrat.c
extern WORD _dos_setumblink (BOOL);                          // memstrat.c

extern void _dos_getbreakflag (BYTE *);                      // break.c
extern void _dos_setbreakflag (BYTE);                        // break.c

WORD _dos_special_allocmem (WORD size, _segment base, _segment ceiling,
                            _segment *baseresult, WORD *sizeresult)
{
    WORD handle;
    register WORD errorcode;
    BYTE oldbreakflag;
    BOOL oldumblink;
    WORD temp;

    if (_osmajor >= 5) {
         _dos_getumblink (&oldumblink);
         if (NOT oldumblink) {
              _dos_getbreakflag (&oldbreakflag);
              if (oldbreakflag) _dos_setbreakflag (OFF);
              _dos_setumblink (ON);
         }
    }

    /*   Enforce 0x0000 <= base <= base + size <= ceiling <= 0xFFFF for the
         block to allocate. */

    if (0xFFFF - base < size) size = 0xFFFF - base;
    if (ceiling < base) ceiling = base + size;
    else if (ceiling - base < size) size = ceiling - base;

    errorcode = MaskMemoryRegion (base, ceiling, &handle);

    if (errorcode == 0) {
         errorcode = _dos_allocmem (size, &temp);
         if (errorcode == 0) {
              *baseresult = temp;
              *sizeresult = size;
         }
         else {
              *sizeresult = temp;
         }
    }
    if (_osmajor >= 5) {
          if (NOT oldumblink) _dos_setumblink (oldumblink);
          if (oldbreakflag) _dos_setbreakflag (oldbreakflag);
    }

    ReleaseMask (&handle);

    return (errorcode);
}
```

MEMBLOCK.C is the memory block management engine for this program and for the SPOTLOAD program in Chapter 5. It allocates all DOS memory blocks, building a linked list of descriptors on the near heap. It may then release all the memory within given bounds, returning a handle (the address of the first descriptor in the linked list) to the caller. In the case of MALLOC.C, the idea is that while all memory outside the desired range is allocated, a normal int 21h function 48h gets memory from within the desired range, according to the prevailing memory strategy, with the least fuss. MALLOC.C releases its "handle" on completion. For the SPOTLOAD program, int 21h function 4B01h will load a program into the desired range, again according to the prevailing strategy, then all other memory may be released before giving the child control; this solves a deficiency in the QEMM LOADHI program, whereby the child cannot allocate any other memory.

```
/* MEMBLOCK.C */

#include    <dos.h>
#include    <malloc.h>
#include    <stdlib.h>

#include    <doserror.h>
#include    <standard.h>

struct MEMBLOCK {
    struct MEMBLOCK *pNext;      // for list management only
    _segment spBase;
    _segment spCeiling;
};

WORD MaskMemoryRegion (_segment, _segment, struct MEMBLOCK **);
void ReleaseMask (struct MEMBLOCK **);

static struct MEMBLOCK *AddBlock (_segment, _segment);
static void RemoveBlock (struct MEMBLOCK *);
static struct MEMBLOCK *pLowestBlock;

static WORD AddLargestBlock (void);
static WORD PunchHole (_segment, _segment);

WORD MaskMemoryRegion (_segment base, _segment ceiling,
                       struct MEMBLOCK **handleptr)
{
    register WORD errorcode;
    /*   If the range is trivial, return immediately. */

    if (base == 0x0000 AND ceiling == 0xFFFF) {
          *handleptr = NULL;
          return (0);
    }

    pLowestBlock = NULL;
    do {
          errorcode = AddLargestBlock ();
    } while (errorcode == 0);

    if (errorcode == E_OUT_OF_MEMORY) errorcode = PunchHole (base, ceiling);

    if (errorcode != 0) ReleaseMask (&pLowestBlock);
    *handleptr = pLowestBlock;
    return (errorcode);
}

static WORD PunchHole (_segment base, _segment ceiling)
{
    register struct MEMBLOCK *ptr;
    register WORD errorcode = 0;
    WORD size;
    struct MEMBLOCK dummy;

    for (ptr = pLowestBlock; ptr != NULL; ptr = ptr -> pNext) {

          /*  Skip blocks that lie completely outside the "hole." Since
              the list of MEMBLOCK structures is sorted by increasing base
              address, the search may be abandoned once the base address
              exceeds the upper limit of the desired region. */

          if (ptr -> spCeiling < base) continue;
          if (ptr -> spBase > ceiling) break;
```

```
        if (ptr -> spBase < base OR ptr -> spCeiling > ceiling) {

            if (ptr -> spBase < base) size = base - ptr -> spBase - 1;
            else size = ceiling - ptr -> spBase;

            errorcode = _dos_setblock (size, ptr -> spBase, &size);
            if (errorcode != 0) break;

            /*   Adjust the corresponding MEMBLOCK structure. */
            ptr -> spCeiling = ptr -> spBase + size;

            errorcode = AddLargestBlock ();
            if (errorcode != 0) break;

            if (ptr -> spBase < base) continue;
        }

        errorcode = _dos_freemem (ptr -> spBase);
        if (errorcode != 0) break;

        dummy.pNext = ptr -> pNext;
        RemoveBlock (ptr);
        ptr = &dummy;
    }

    return (errorcode);
}

static struct MEMBLOCK *AddBlock (_segment base, _segment ceiling)
{
    register struct MEMBLOCK **mark, *ptr;
    struct MEMBLOCK *new;

    new = malloc (sizeof (struct MEMBLOCK));
    if (new == NULL) return (NULL);

    mark = &pLowestBlock;
    ptr = pLowestBlock;
    while (ptr != NULL AND ptr -> spBase <= base) {
        mark = &(ptr -> pNext);
        ptr = ptr -> pNext;
    }
    new -> pNext = ptr;
    new -> spBase = base;
    new -> spCeiling = ceiling;
    *mark = new;

    return (new);
}

static void RemoveBlock (struct MEMBLOCK *memblock)
{
    register struct MEMBLOCK **mark, *ptr;

    mark = &pLowestBlock;
    ptr = pLowestBlock;
    while (ptr != NULL AND ptr != memblock) {
          mark = &(ptr -> pNext);
          ptr = ptr -> pNext;
    }
    if (ptr == memblock) {
          *mark = ptr -> pNext;
          free (ptr);
    }
}

static WORD AddLargestBlock (void)
{
    register struct MEMBLOCK *memblock;
    register WORD errorcode;
    WORD base;
    WORD size;

    /*    Obtain the largest free block available from DOS. */
    errorcode = _dos_allocmem (0xFFFF, &size);
    if (errorcode == 0) {
          base = size;
          size = 0xFFFF;
    }
    else if (errorcode == E_OUT_OF_MEMORY)
          errorcode = _dos_allocmem (size, &base);

    if (errorcode == 0) {
          memblock = AddBlock (base, base + size);
          if (memblock == NULL) errorcode = E_ACCESS_DENIED;
    }
    return (errorcode);
```

```
}

void ReleaseMask (register struct MEMBLOCK **starter)
{
    register struct MEMBLOCK *ptr;

    ptr = *starter;
    while (ptr != NULL) {
          *starter = ptr -> pNext;
          _dos_freemem (ptr -> spBase);
          free (ptr);
          ptr = *starter;
    }
}
```

EATBLOCK.C is the top-level module for the EATBLOCK program, containing its command-line processing. The `main` function in EATBLOCK.C calls the `_dos_special_allocmem` function in MALLOC.C, and then either calls `AssignBlock` (for EATBLOCK /p) or `RunTestShell`, both of which are also contained in EATBLOCK.C.

```
/* EATBLOCK.C - top-level module for EATBLOCK.EXE */

#include     <dos.h>
#include     <errno.h>
#include     <process.h>
#include     <stdarg.h>
#include     <stdio.h>
#include     <stdlib.h>
#include     <string.h>

#include     <arena.h>
#include     <doserror.h>
#include     <standard.h>

char description [] = "\n\
Allocates a memory block specified by both size and location.\n\
\n\
  EATBLOCK size [base [ceiling]] [/p]\n\
\n\
All numeric values are interpreted as hexadecimal. Size is in paragraphs.\n\
Base and ceiling are segment addresses.\n\
\n\
Use size=FFFF to report on the largest free block within the given range.\n\
\n\
Ordinarily, EATBLOCK starts a command processor to support testing under the\n\
new memory environment. The memory block is released when the command\n\
processor exits. The /p switch instructs EATBLOCK to make its allocation\n\
permanent and to exit immediately (i.e., without starting a shell).\n";

/*  Subfunctions to separate these cases (with and without /p, respectively)
    and a string for use when starting a command shell   */

static int AssignBlock (_segment);
static int RunTestShell (void);

static char instructions [] = "\n\
A command processor is being started to allow investigation with\n\
whatever tools you prefer. Type EXIT when satisfied.\n";

/*  Special memory allocation function - similar to the Microsoft C library
    function _dos_allocmem, but with extra arguments to specify range and
    passing separate pointers for base and size results. The function is
    defined and annotated in the file MALLOC.C. */

extern WORD _dos_special_allocmem (WORD, _segment, _segment,
                                   _segment *, WORD *);
/*  Miscellany  */

#define arenaptr(seg)        ((struct ARENA_HEADER _far *) MK_FP (seg, NULL))

extern BOOL ReadHexWord (char *, WORD *);          // in readhex.c
static void PutError (char *, ...);

/*  ====================================================================  */

int main (int argc, char **argv)
{
    char *ptr;
    WORD ch, errorcode, size, temp;
    _segment base = 0x0000, ceiling = 0xFFFF;
    BOOL sizedone = FALSE, basedone = FALSE, ceilingdone = FALSE;
    BOOL permanent = FALSE;

    /*   Parse the command line. */

    while (++ argv, -- argc) {
         ptr = *argv;
```

```
        if (*ptr == '/') {
            if (ptr [2] == '\0') {
                ch = ptr [1] | 0x20;
                if (ch == '?') {
                    printf (description);
                    return (0);
                }
                else if (ch == 'p') {
                    permanent = TRUE;
                    continue;
                }
            }
            PutError ("invalid switch %s", ptr);
            return (-1);
        }
        else {
            if (NOT ReadHexWord (ptr, &temp)) {
                PutError ("invalid parameter %s", ptr);
                return (-1);
            }
            if (NOT sizedone) {
                size = temp;
                sizedone = TRUE;
            }
            else if (NOT basedone) {
                base = temp;
                basedone = TRUE;
            }
            else if (NOT ceilingdone) {
                ceiling = temp;
                ceilingdone = TRUE;
            }
            else {
                PutError ("too many parameters %s", ptr);
                return (-1);
            }
        }
    }
    if (NOT sizedone) {
        printf (description);
        return (0);
    }

    /*  Attempt to allocate a memory block of the desired size within the
        specified range. */
    errorcode = _dos_special_allocmem (size, base, ceiling, &base, &size);
    if (errorcode != 0) {

        /*  All forms of allocation error are terminal. */

        if (errorcode == E_OUT_OF_MEMORY) {
            /*   The usual reason for failure is simply that no free memory
                 block meets the requirements, in which case, report the size
                 of the largest suitable block.        */

            printf ("\nLargest block available between segments %04Xh "
                    "and %04Xh is %04Xh paragraphs\n", base, ceiling, size);
        }
        else if (errorcode == E_ACCESS_DENIED) {
            /*   Another cause is that the _dos_special_allocmem function
                 found that it could not hold information about memory blocks
                 outside the requested range while forcing DOS to allocate
                 from within the range.      */

            PutError ("too much memory fragmentation, too little heap space");
        }
        else {
            /*   Any other error condition almost certainly indicates that
                 the memory control chain is corrupt. */

            PutError ("serious memory control problem");
        }
        return (-1);
    }
    /*   Having "eaten" a suitable block of memory, affirm the success, then
         either assign the block permanently and exit or locate the command
         processor and run it as a test shell. */

    printf ("\nAllocated %04Xh paragraphs at segment %04Xh\n", size, base);
    return (permanent ? AssignBlock (base) : RunTestShell ());
}

static int RunTestShell (void)
{
    register char *pCOMSPEC;
    int exitcode;
    /*   Consult the environment for the command processor's pathname,
         display an explanatory message and then run the shell (with an
         empty command line). */

    pCOMSPEC = getenv ("COMSPEC");
    if (pCOMSPEC == NULL) {
         PutError ("cannot find command processor for test");
         return (-1);
    }
    printf (instructions);
    exitcode = spawnl (P_WAIT, pCOMSPEC, pCOMSPEC, NULL);

    if (exitcode == -1) PutError ("%s", strerror (errno));
    else {
         /*  On leaving the command processor, remind the user that the shell
             was executed as part of a memory allocation test - just for
             clarity, it being far too easy to get lost when multiple copies
             of the command processor are run. */

         printf ("\nEATBLOCK:  memory allocation test completed.\n");
    }
    return (exitcode);
}

#pragma optimize ("a", on)

static int AssignBlock (_segment base)
{
    /*   Ordinarily, the block will be released when the program terminates
         (without intending to stay resident). This may be prevented by
         reassigning the block's ownership beforehand.

         Incidentally, without the /Oa option (forced by the optimize pragma
         preceding the function), the compiler has little choice but to
         recalculate base - 1 in each of the following lines, since no matter
         how improbably, the assignments could conceivably alter the value of
         base from line to line. */

    arenaptr (base - 1) -> cName [0] = 'S';           // system
    arenaptr (base - 1) -> cName [1] = 'C';           // code
    arenaptr (base - 1) -> spOwner = ARENA_DOS;          // 0008h
    return (0);
}
#pragma optimize ("a", )
```

```
static void PutError (char *str, ...)
{
    va_list va_start (argptr, str);
    printf ("\nEATBLOCK fatal error:  ");
    vprintf (str, argptr);
    printf ("\n");
    va_end (argptr);
}
```

READHEX.C contains a function which, not surprisingly, reads hexadecimal numbers.

```
/* READHEX.C */

#include    <standard.h>

BOOL ReadHexWord (register char *ptr, WORD *result)
{
    register WORD ch;
    WORD temp = 0;

    ch = *ptr;                                  // be kind to users
    if (ch == '0' OR ch == '\\') {              // accustomed to C-style
         if ((ptr [1] & ~0x20) == 'X') ptr += 2; // expressions beginning
    }                                           // with "\x" or "0x"

    while (*ptr == '0') ptr ++;      // skip leading zeros

    while (ch = *ptr ++) {
         if (ch >= '0' AND ch <= '9') ch -= '0';       // decimal digit
         else {
             ch &= ~0x20;
             if (ch >= 'A' AND ch <= 'F') ch -= ('A' - 10);  // hex digit
             else if (ch == 'H') break;                      // final 'h'
             else return (FALSE);
         }
         if (temp & 0xF000) return (FALSE);     // accumulate result
         temp <<= 4;                            // in temp, guarding
         temp += ch;                           // against overflow
    }
    *result = temp;
    return (TRUE);
}
```

BREAK.C contains two simple C wrappers around int 21h function 33h to get and set the DOS break flag.

```
/* BREAK.C */

#include    <standard.h>

void _dos_getbreakflag (BYTE *breakflagptr)
{
    BYTE breakflag;
    _asm {
            mov     ax,3300h
            int     21h
            mov     breakflag,dl
    }
    *breakflagptr = breakflag;
}

void _dos_setbreakflag (BYTE breakflag)
{
    _asm {
            mov     dl,breakflag
            mov     ax,3301h
            int     21h
    }
}
```

There's not enough complexity here to justify presenting the obvious makefile. The only point to make, really, is the desirability of setting a low maxalloc value either at link time or with EXEHDR.

Memory allocated with EATBLOCK shows up with MEM /P. For example, after running eatblock 16 2000 3000 /p, mem /p displays

```
01FFF0      MSDOS       000160     System Program
```

The 01FFF0 address shown by MEM is the location of the newly-eaten block's MCB one paragraph below the requested location of 2000h. MEM indicates the block is a "System Program" because of the "instant TSR" method used by the `AssignBlock` function in EATBLOCK.C above.

The completed EATBLOCK program is not just a diagnostic tool to consume memory so that other programs may have their operations tested under a variety of conditions. More than this, it allows the easy construction of memory blocks at precise locations for more elaborate purposes (but which do not justify writing a dedicated program).

Consider, for instance, how one might demonstrate the previous chapter's claim that the first set of SFTs built outside the DOS kernel must lie in low memory lest Windows fail to run. The bold way is to move the existing SFTs to upper memory (updating the SFT header chain appropriately) and then to start Windows enhanced mode. Be sure, however, that the first 512K of memory cannot (by chance) contain three equally-spaced copies of the characters "CON".

Begin by running **mem /d**, noting the size set aside for SFTs (marked as FILES= by MEM) in the main configuration block:

```
002640      IO          0048E0      System Data
              HIMEM       0004A0       DEVICE=
                XMSXXXX0                 Installed Device Driver
              XMSWATCH    0009E0       DEVICE=
                XMSWWWW0                 Installed Device Driver
              EMM386      001AC0       DEVICE=
                EMMXXXX0                 Installed Device Driver
                          000820       FILES=
                          000050       FCBS=
                          000200       BUFFERS=
                          0008F0       LASTDRIVE=
                          000BC0       STACKS=
```

Thus, 82h paragraphs are required in upper memory. EATBLOCK will find a suitable block of memory, if one is available:

```
F:\>eatblock 0082h A000h /p

Allocated 0082h paragraphs at segment DBD0h
```

Now start a debugger, preferably one like SYMDEB or even CodeView that has flexible commands for dumping and editing dwords. Starting from the address of the SYSVARS structure within the DOS kernel (which will be familiar to the reader if ever any system-level inspection has been required for the machine involved in this exercise), it is harder to describe than to actually take the following steps: Find the second set of SFTs, verify that there are no further sets, copy the second set to the area obtained by EATBLOCK, and link the first set to the new one in upper memory.

```
-dd 0123:002a l1
0123:002A  0123:00CC
-dd 0123:00cc l1
0123:00CC  04FD:0000
-dd 04FD:0000 l1
04FD:0000  0000:FFFF
```

```
-m 04fd:0000 10820 dbd0:0000
-ed 0123:00cc dbd0:0000
-q
```

Were this not being attempted under test conditions, it would be necessary to disable interrupts while copying the SFTs and loading the far pointers. That aside, the system now has five SFTs in the DOS kernel and some more in upper memory—just as if DOS had built them there in the first place.

Now run Windows enhanced mode. To find the size of the SFT structure, Windows opens the CON device five times; if there are only two SFT blocks, then three of the SFTs created for the CON device must lie within one or other of these blocks, hopefully in sequence. Using EATBLOCK and SYMDEB, we have moved the second SFT to upper memory, where Windows fails to search for the "CON" strings. The result is the following message:

```
ERROR: Unsupported MS-DOS version.
```

It may never be known whether or not DOS 5 builds the configuration structures for the **buffers**, **fcbs**, **files**, **lastdrive,** and **stacks** commands in low memory rather than upper memory merely to avoid this potential conflict with Windows, but at least the problem can be demonstrated with reasonable ease.

Walking the Memory Arena Chain

The following code will not actually be given a practical use until Part 3, when the XMS memory mapping utility (which monitors communications between the XMS server and its clients) needs to identify the particular client asking for an extended memory allocation. However, the code itself fits into this discussion of DOS memory management.

In addition to some lower-level arena management functions, ARENA.C contains a set of functions that concentrate on the arena chain as a whole or on memory block ownership. For example, `GetArenaSegment` finds the segment address of the arena header for the memory block containing a given segment address, and `GetSubArenaSegment` performs the same operation for segments owned by DOS that have subarenas:

```
/* ARENA.C */

#include     <arena.h>
#include     <psp.h>
#include     <standard.h>
#include     <sysvars.h>

/*  The module requires the address of the DOS kernel's SYSVARS structure in
    order to locate the first memory arena header. */

extern struct SYSVARS _far *lpSysvars;
```

```
/*  By defining the symbol _MANUAL_UMBLINK_ at compile time and initializing
    the following pointer at run-time, this module's search of the memory
    arena system extends to the upper-memory chain without requiring
    explicit UMB linkage through a DOS function. This allows a search of
    all DOS memory to be conducted (by the GetArenaSegment function) in
    circumstances where DOS functions cannot be used (e.g., from within a
    device driver call). */

#ifdef _MANUAL_UMBLINK_
extern WORD _far *lpUMBLinkSeg;
#endif

/*  In the situation where this module is imagined to be needed while DOS
    calls are not available, the default data segments in DGROUP and the
    default code segment _TEXT will have been discarded in favor of keeping
    only a very small resident footprint for the program. In this case, the
    variables _osmajor and _osminor in which the C run-time records the DOS
    version for later reference will no longer be present:  initialization
    code for this module must record the version number at a location
    guaranteed to remain part of the program. */

#ifdef _MANUAL_UMBLINK_
extern struct {
    BYTE minor;
    BYTE major;
} wDOSVersion;
#define _osmajor wDOSVersion.major
#define _osminor wDOSVersion.minor
#else
extern BYTE _osminor;                           // defined in STDLIB.H
extern BYTE _osmajor;                           // defined in STDLIB.H
#endif

/*  Macros to create far pointers to memory arena headers and PSPs from just
    the segment addresses  */

#define MK_ARENA_FP(seg)    ((struct ARENA_HEADER _far *) MK_FP (seg, 0))
#define MK_PSP_FP(seg)           ((struct PSP _far *) MK_FP (seg, 0))

/*  ------------------------------------------------------------------------   */

/*  Begin with two short routines for basic arena header management. The
    first function extracts an arena header's signature byte after checking
    its validity. */
```

```
BYTE GetValidArenaID (_segment arenaseg)
{
    BYTE id = MK_ARENA_FP (arenaseg) -> cSignature;
    if (id == ARENA_IN_CHAIN OR id == ARENA_LAST) return (id);
    else return (0);
}

/*  The second function takes the segment address of a supposed arena header
    and returns the segment address of the corresponding memory block's
    upper bound - where there will usually be the next arena header. */

_segment GetNextArena (_segment arenaseg)
{
    register WORD inc;

    /*    The distance (in paragraphs) to the next header is the size of the
          memory block which follows the header, plus one more paragraph for
          the header itself. */

    inc = MK_ARENA_FP (arenaseg) -> wParagraphs + 1;

    /*    Before adding this increment to the given arena segment, consider
          that two pathological cases may occur if the arena header is
          corrupt (or if the segment does not in fact address a true arena
          header).

          The main problem occurs if the number of paragraphs in the block is
          recorded as 0xFFFF, giving an increment of zero. This case is
          dangerous, since its omission while following an arena chain (or a
          supposed one) results in an infinite loop.

          Secondly, the addition may overflow the available 16 bits (the
          memory arena chain does not wrap around the 1MB boundary, so neither
          should the calculation). In a routine that "walks" the arena chain,
          this case would probably be caught as an invalid signature on the
          next round and is arguably not so important.

          Incidentally, DOS checks neither condition when it examines arena
          headers. */

    if (inc == 0 OR inc > 0xFFFF - arenaseg) return (0);
    return (arenaseg + inc);
}
```

```
/* ---------------------------------------------------------------------- */

/* At the next level are functions which concentrate on the larger scale of
   the arena chain or on memory block ownership. The simplest example is a
   function which tests whether a given segment is plausibly a memory arena
   header by checking that it is followed by a well-defined chain. This
   function does nothing else along the way;  note though, that the two
   earlier functions spare this one from bothering with the details of
   validation and defective iteration. */

_segment ValidateArenaChain (register _segment seg)
{
    BYTE id;

    do {
        /* At each step, use the functions provided earlier to validate the
           supposed arena header and find the upper bound to the
           corresponding memory block. */

        id = GetValidArenaID (seg);
        if (id == 0) return (0);
        seg = GetNextArena (seg);
        if (seg == 0) return (0);

        /* Continue for as long as there seem to be more headers. */

    } while (id == ARENA_IN_CHAIN);

    /*  Return the chain's upper bound. Thus, for a machine with 640KB of
        conventional memory and no upper memory under DOS's control, the
        function returns 0xA000 if given the segment address of a valid
        arena header in low memory. With upper memory available in a
        dos=umb configuration but with the UMB link disabled, the function
        returns 0x9FFF, it having no knowledge that DOS starts a second
        chain where the first one ends. */

    return (seg);
}

/* The following function finds the segment address of the arena header
   for the memory block containing a given segment address. */

_segment GetArenaSegment (_segment givenseg)
{
    register _segment seg;
```

```
_segment markseg;
BYTE id;

/*   In DOS versions 5 and higher, there may be two chains of memory
     arena headers. The DOS kernel has a variable in which to keep the
     segment address of the header at the boundary. The address of this
     kernel variable (if it exists) must be recorded by initialization
     code (in another module) before using this function. */

#ifdef _MANUAL_UMBLINK_
_segment umblinkseg = lpUMBLinkSeg != NULL ? *lpUMBLinkSeg : 0xFFFF;
#endif

/*   Seed the search with the segment address of the first memory arena
     header - available as the word before the SYSVARS structure in the
     DOS kernel's data area. Then make sure that the paragraph address
     given as the function's argument does not lie below DOS's memory
     control system. */

seg = *((WORD _far *) lpSysvars - 1);
if (givenseg < seg) return (0);

/*   Follow the chain of memory arena headers upward through memory
     until either the desired segment is passed or the chain is
     exhausted. */

for (;;) {

     markseg = seg;

     /* At each step, validate the supposed arena header and find the
        upper bound to the corresponding memory block. */

     id = GetValidArenaID (seg);
     if (id == 0) return (0);
     seg = GetNextArena (seg);
     if (seg == 0) return (0);

     /* If markseg <= givenseg < seg, then the desired block has been
        found. The first half of this inequality is already established
        (either from the preceding step or as an entry condition for the
        iteration). */

     if (givenseg < seg) break;
```

```
        /*  If no more blocks exist in the chain, then abort the search. In
            DOS 5 and higher, a second chain may start immediately after the
            first. The _MANUAL_UMBLINK_ symbol supports examination of the
            second chain in cases where explicit linkage through int 21h
            function 5802h is not possible. */

        if (id == ARENA_LAST) {
            #ifndef _MANUAL_UMBLINK_
            return (0);
            #else
            if (umblinkseg == 0xFFFF OR umblinkseg != seg) return (0);
            #endif
        }
    }

    /*  If the block containing the given segment is not the last in the
        chain, check that the remainder of the arena chain is plausible
        before returning the result. */

    if (id == ARENA_IN_CHAIN) {
        if (ValidateArenaChain (seg) == 0) return (0);
    }
    return (markseg);
}

/*  ---------------------------------------------------------------------  */

/*  As an aside, observe that the memory arena chain may conceivably provide
    the most reliable way to determine whether code is being executed during
    device driver initialization. Of the various traces left by the
    temporary PSP used by SYSINIT when building the system (and particularly
    while loading device drivers), the existence of a memory block owned by
    the temporary PSP is fairly secure against interference from
    user-supplied code. */

static BOOL TestSysinitPSP (_segment seg)
{
    /*  If the segment does not look like it addresses a PSP, then fail. */

    if (*((WORD _far *) MK_PSP_FP (seg) -> Int20h) != 0x20CD) return (FALSE);

    /*  The SYSINIT PSP is best distinguished by having no parent. Fail the
        function if the segment addresses an ordinary PSP. */

    if (MK_PSP_FP (seg) -> spParentPSP != 0) return (FALSE);
    /*   Otherwise, return TRUE - not as strict affirmation that the segment
         addresses the SYSINIT PSP, but that this identification is not
         implausible. */

    return (TRUE);
}

static _segment GetSysinitPSP (void)
{
    register _segment seg, ownerseg;
    BYTE id;

    /*   Search the arena chain in conventional memory for a block assigned
         to the SYSINIT PSP. In versions of DOS up to and including 5.0,
         the block of low memory into which SYSINIT loads device drivers and
         builds tables is not assigned to owner = 0x0008 until the system is
         formally ready to execute programs. */

    seg = *((WORD _far *) lpSysvars - 1);
    do {
         id = GetValidArenaID (seg);
         if (id == 0) break;

         ownerseg = MK_ARENA_FP (seg) -> spOwner;
         if (TestSysinitPSP (ownerseg)) return (ownerseg);

         seg = GetNextArena (seg);
         if (seg == 0) break;

    } while (id == ARENA_IN_CHAIN);
    return (0);
}

#define IsInConfig()         (GetSysinitPSP () != 0 ? TRUE : FALSE)

/*  ====================================================================  */

static BOOL SupportsSubArenas (_segment seg)
{
    register _segment ownerseg;

    /*   The subarena scheme for detailing memory allocations made by DOS for
         device drivers and kernel structures such as SFTs, CDSs, etc. was
         introduced with DOS 4. */
    if (_osmajor < 4) return (FALSE);

    /*  In general, the only memory blocks subdivided this way have owner
        fields indicating their private use by DOS (owner == 0x0008). */

    ownerseg = MK_ARENA_FP (seg) -> spOwner;
    if (ownerseg == ARENA_DOS) {

        /* Additionally, in DOS 5 and higher, the letters 'SD' (for System
           Data) must start the name field. */

        if (_osmajor >= 5) {
           #define DEREF_WORD(fp) *((WORD _far *) (fp))
           if (DEREF_WORD (MK_ARENA_FP (seg) -> cName) != MK_WORD ('D', 'S'))
                return (FALSE);
        }
    }
    else {

        /* There is a special case, however, which must be considered when
           the subdivision of a memory block is examined during device
           driver initialization. While SYSINIT is still building drivers
           and tables into a memory block, ownership lies with SYSINIT's
           temporary PSP and is not assigned the special value 0x0008 until
           later. */

        if (NOT TestSysinitPSP (ownerseg)) return (FALSE);
    }
    return (TRUE);
}

static char ValidSubArenaIDs [] = {
    'B',        // buffers
    'D',        // device
    'E',        // buffers /x (EMS hook and driver dummy,
                   //   plus hook for int 2Fh function 1Bh - DOS 4 only)
    'F',        // files (SFTs)
    'I',        // ifs (DOS 4 only)
    'L',        // lastdrive (CDSs)
    'S',        // stacks
    'T',        // install (temporary code)
    'X'         // fcbs (FCB-SFTs)
};
```

```
BYTE GetValidSubArenaID (_segment subarenaseg)
{
    register BYTE *ptr;
    register WORD count;
    BYTE ch, id;

    /*  Simply check the ID against a list of values known to be used.    */

    id = MK_ARENA_FP (subarenaseg) -> cSignature;
    ptr = ValidSubArenaIDs;
    count = sizeof (ValidSubArenaIDs) / sizeof (ValidSubArenaIDs [0]);
    do {
         ch = *ptr ++;
         if (ch == id) return (ch);
    } while (-- count);
    return (0);
}

_segment GetSubArenaSegment (_segment baseseg, _segment givenseg)
{
    register _segment seg;
    _segment markseg, topseg;
    BYTE id;

    /*  Begin by checking that the arena header at the segment address given
         by the argument baseseg supports a chain of subarena headers. */

    if (NOT SupportsSubArenas (baseseg)) return (0);

    /*  The subarena headers should lie within the block of memory described
         by the main header. */

    topseg = GetNextArena (baseseg);
    if (topseg == 0) return (0);

    /*  The subheaders begin at the paragraph immediately after the main
         arena header. */

    seg = baseseg + 1;
    markseg = 0;

    /*  Follow the chain of memory arena headers upward through memory
         until either the desired segment is passed or the chain is
         exhausted. */
    for (;;) {

          id = GetValidSubArenaID (seg);
          if (id == 0) break;

          /*  Having encountered a plausible subarena header, record its
              segment address, since it may be the one that is wanted. */

          markseg = seg;

          /*  Find the end of the sub-block. Stop looping if this overruns
              the main block (which the subarenas are supposed to subdivide). */

          seg = GetNextArena (seg);
          if (seg == 0) break;
          if (seg >= topseg) break;

          /*  If markseg <= givenseg < seg, then the desired sub-block has
              been found. Otherwise, continue the search. */

          if (givenseg < seg) return (markseg);
    }

    /*    There is a circumstance in which the chain of subarena headers may
          overrun the upper bound obtained from the main block and not
          indicate corruption. When device drivers are initialized, their
          subarena header is only partially valid. The size field is
          untouched until the device driver's initialization is complete.
          This means that following a subarena chain at this stage of the
          system's startup sequence is technically unreliable. Defensive
          coding should prevent mishap and reduce the opportunity for spurious
          identification, but the latter cannot be elimated entirely. */

    if (NOT IsInConfig ()) return (0);
    else return (markseg);
}
```

With its high ratio of comments to actual code, this ARENA module should be self-explanatory.

CHAPTER 5

Program Execution

It is well known that DOS programs are launched with int 21h function 4Bh (EXEC). Usually subfunction 0 is used to both load and execute a program. However, there are other subfunctions, including a formerly-undocumented subfunction 1 to load (but not execute) a program, and subfunction 5 (first introduced in DOS 5) for Set Execution State. This chapter explains the importance of subfunction 5 and presents a SPOTLOAD program that loads a program into a specified memory region (the effect, though not the implementation, is somewhat similar to the LOADHIGH /L option introduced in DOS 6).

Table 5-1: DOS function 4Bh—Load and/or Execute Program/Overlay (EXEC)

INPUT:

AH 4Bh

AL subfunction

00h	load and execute program
01h	load program
03h	load overlay

DS:DX address of program/overlay pathname as ASCIIZ string

ES:BX functions 4B00h and 4B01h: address of parameter block

OFFSET	SIZE	DESCRIPTION
00h	word	segment address of environment
02h	dword	address of command line
06h	dword	address of first FCB
0Ah	dword	address of second FCB
0Eh	dword	for function 4B01h only—will be filled with address of initial stack
12h	dword	for function 4B01h only—will be filled with address of entry point

Table 5-1: Continued

function 4B03h: address of parameter block

OFFSET	SIZE	DESCRIPTION
00h	word	segment address at which to load overlay
02h	word	segment address from which to calculate relocations

SUCCESS (AFTER CHILD PROCESS HAS TERMINATED):

flags	all defined flags are clear, except that interrupts are enabled
ax	corrupt

FAILURE:

cf	set	
ax	0001h	if AL = 02h, 04h or is > 05h
	0002h	if attempt to open file fails because file is not found; or if pathname describes a character device
	0003h	if attempt to open file fails because path is not found
	0004h	if attempt to open file fails because too many files are already open
	0005h	if attempt to open file fails for any reason not covered elsewhere
	0008h	for any memory allocation error; or if available memory does not meet specification of .EXE header
	000Ah	if environment exceeds 32K
	000Bh	for any read error

CORRUPTS:

bx

dx

The exec function is a complicated piece of code. It seems best to follow the function through its operations:

The first task is to open the file described by the pathname at DS:DX. If the file cannot be opened then obviously its contents cannot be executed. To open the file the kernel does not issue an int 21h function 3Dh (int 21h is not reentrant), but does call the procedure that would normally handle function 3Dh. Only read access is required and no file sharing restrictions are requested.

DOS 5 and 6, but not earlier versions, support a redirector's possible need to distinguish the case of a file being opened specifically for execution. Thus, if REDIR is installed and the pathname refers to a remote drive, DOS passes the mode 23h when it hands the Open File request to the redirector with the appropriate int 2Fh function 11h call. The value 3 in this mode is disallowed in all other circumstances. The 2 is just the natural precaution to deny shared write access, in case a process interrupts the exec function and attempts to open and write to this same file.

DOS fails the function if the string at DS:DX turns out to describe a character device. To distinguish a character device from a disk file, the DOS kernel calls the procedure that would normally handle function 4400h (IOCTL Get Device Information).

Two considerations arise from the way that int 21h function 4Bh is handled by calling internal procedures also responsible for other DOS functions. One is that, like any well-written program, int 21h function 4Bh must prepare for the possibility that one of these other DOS functions might encounter a DOS exception (though only critical errors are likely to be relevant here) and therefore not return as expected. This has in fact been well-guarded by the DOS kernel (save only for a very slight omission), as will be seen in detail later.

The other consideration is that between calling the various internal procedures corresponding to DOS functions, the kernel is liable to be interrupted and re-entered. The concern here is not for TSRs following normal programming techniques: These may well have hooked a hardware interrupt (timer or keyboard) and therefore break into the processing of int 21h function 4Bh, but they will notice that the InDOS flag is set and thence decline to use int 21h during their "pop-up" stage. Instead, the problem lies with more adventurous software attempting to reproduce the known behavior of network servers and Windows enhanced mode, which is to interrupt the DOS kernel irrespective of the InDOS flag's value, preferring to note instead whether the DOS kernel has declared itself (with an int 2Ah call) to be executing a "critical section" of code. As long as the kernel is not in one of these critical sections, it may be interrupted and re-entered, providing the interrupter saves and restores the contents of the Swappable Data Area (SDA) and other instance regions.

As so often happens with the word "other," it is much more significant here than it may seem. The SDA appears to have been defined to meet the needs of a network server. For fast response to requests from clients for file I/O on the server, it is desirable that the network software be able to interrupt the DOS kernel and issue int 21h instructions even if the kernel is already processing int 21h instructions. Performance would be very poor if the network had to poll the InDOS flag like an ordinary TSR. Note, however, that running programs in the server's address space is not envisaged as a network operation (instead, a client opens and reads a file across the network so that the file's contents may be executed on the client's CPU). The code that handles int 21h function 4Bh does not expect to be re-entered by a network server and uses several items of kernel data which do *not* lie within the SDA. This is a problem for the adventurous task-switcher or pop-up TSR. These programs may locate the SDA easily enough because an albeit-undocumented

DOS function (int 21h function 5D06h) delineates the region. But what can such software do about the other data items?

These are points to bear in mind as the execution sequence unfolds. The very next step in that sequence, having established that the executable file exists, is to set up the new process's environment.

The Program Environment

For programs, but not overlays, DOS provides an environment—a separate memory allocation in which the parent process may pass a set of strings to the child it is creating with function 4Bh. Given a default first-fit memory allocation strategy the environment block will lie below the program itself, as memory for the environment is sought first. DOS caters for three possibilities:

- The parameter block at ES:BX may provide the segment address of an environment block already prepared for the child and which need simply be copied when DOS is ready.
- An easier alternative is that the parent leaves the value 0000h in the parameter block's environment field (offset 0) as a direction to copy the child's environment directly from the parent's.
- The final possibility is that the child should have no environment at all, which is what happens if the environment field in the parameter block contains 0000h but the parent has no environment to copy.

If an environment is wanted (and note that too few programs seem to be aware that they can be executed without one), DOS scans the source environment to ensure that it does contain a set of null-terminated strings. The set itself must be terminated by a null string. If DOS cannot find the end within 32K, it fails the function. The child environment will receive a copy of these strings, followed by the word 0001h and a copy of the program's pathname supplied by the parent at DS:DX. Note the implication: Programs that inspect their environment to determine the directory from which they have been launched must not assume that the pathname in the environment has been qualified, even partially.

For an example of a program that makes this assumption, look no further than the MS-DOS Shell distributed with DOS 5 or 6. Change to the drive and directory containing the various DOSSHELL programs, but instead of running the shell from the COMMAND prompt, launch it from DEBUG or SYMDEB. For example

```
C:\DOS>debug dosshell.com
-g
```

The program will run, but not as expected if the DOSSHELL.INI settings were ever changed. For instance, if the Shell is ordinarily configured for a graphics mode, it will now run in the default 25-line text mode with the default color scheme and, indeed, all the default options: DOSSHELL has been unable to find the DOSSHELL.INI file in the same directory as the executables. The difference in behavior is simply because DEBUG and SYMDEB, unlike COMMAND.COM, do not parse program names before calling int 21h function 4B01h to load the program being debugged. Supplying the full pathname before presentation to DEBUG, as in **debug c:\dos\dosshell.com**, causes the DOS Shell to work as expected.

Another readily available example is the WIN.COM program, which loads Windows 3.0 (but not Windows 3.1, in which the oversight has been fixed). The effect for Windows 3.0 is seen easily enough by starting **debug win.com** from the Windows directory and then opening a DOS Prompt in which to enter the **set** command. This should reveal the specially-constructed **windir** environment variable, whose value should be the path to the main Windows directory. WIN.COM deduces this directory from the pathname it finds in its environment. But it *assumes* that this pathname will contain a backslash! If no backslash happens to be present, then WIN.COM may corrupt some of the strings in its own environment and thence those of all its children. The effects on other programs are essentially unpredictable, ranging from absolutely nothing to some real "head-scratchers" (if, for instance, the PATH happened to be the last environment variable and is therefore corrupt the entire time Windows is running). The best thing that can be said about this strikingly simple oversight is that its detection must have been very easy for Microsoft, since the SDK's provision for debugging Windows 3.0 real mode is to run WIN.COM under the control of the SYMDEB debugger. Cynics may like to ponder first whether any Windows developers encountered the problem, second whether the problem would have been reported, and finally whether Microsoft ever explained the very simple cause.

Having decided the size of the child's environment, a suitable memory block is sought straightaway and loaded with the expected data, that is, the environment variables from the designated source, the word 0001h (indicating one pathname follows), and the pathname from DS:DX. Observe that the allocation is subject to the strategy in force when int 21h function 4Bh was called. Failure to obtain memory for the environment block, given that an environment is wanted, is fatal to the function.

Executable File Formats

DOS must now determine whether the program is in .EXE or .COM format. The file extension is irrelevant to this purpose. Instead, DOS depends on the presence in .EXE files of the header structure shown in Table 5-2.

Table 5-2: Header for Programs in .EXE Format

OFFSET	SIZE	DESCRIPTION
00h	word	the letters 'MZ' or 'ZM'
02h	word	number of bytes in last page—*not* used by DOS kernel when loading program or overlay
04h	word	number of 512-byte pages (whole or partial)
06h	word	number of entries in the relocation table
08h	word	number of 16-byte paragraphs in file before start of program material
0Ah	word	minimum number of paragraphs of memory required by program in excess of the image loaded from disk
0Ch	word	maximum number of paragraphs of memory desired by program in excess of the image loaded from disk
0Eh	word	initial stack segment—relative to start of program image
10h	word	initial stack pointer
12h	word	checksum—not used by DOS kernel when loading program
14h	word	initial instruction pointer
16h	word	initial instruction segment—relative to start of program image
18h	word	offset to relocation table from start of header

The structure is 1Ah bytes long, so DOS attempts to read this many bytes from the start of the file into a buffer set aside in the kernel's data segment. This buffer lies outside the Swappable Data Area. Because of this and other similar cases, an adventurous task switcher trying to improve response by techniques that allow interruption while inside DOS must not permit DOS function 4Bh to be reentered. Windows enhanced mode is able to circumvent the problem in DOS 5 and higher because the kernel supplies the address of this buffer as instance data via int 2Fh function 1605h. For earlier versions of DOS, the DOSMGR VxD knows of a sequence of instructions (from the end of the code for the exec function, incidentally) from which it may deduce the location of the buffer.

A read error while attempting to load these first 1Ah bytes of the executable is, of course, fatal to the function; so is a lack of input, that is, attempting to read 1Ah bytes but obtaining none. DOS accepts the program as having .EXE format if all 1Ah bytes are read successfully and the first two bytes match the initials of Mark Zbikowski, 'MZ' (5A4Dh) or 'ZM' (4D5Ah). Otherwise, DOS treats the program as a .COM file (which, because it lacks a header, can be smaller than 1Ah bytes).

Preparing for DOS Exceptions

Consider carefully that when DOS reads from the file, a disk error will generate an int 24h exception. At the time, the DOS kernel will have at least an open file handle and a memory block for the environment. Before long, it will also have a memory block for the program itself. What happens if int 24h doesn't return? It is preferable, of course, that software follow instructions and return from int 24h with one of the following values in register AL:

`00h`	`ignore`
`01h`	`retry`
`02h`	`abort`
`03h`	`fail`

Remember, however, that the int 24h handler is given the DOS caller's stack, as shown in Table 5-3.

Table 5-3: Stack Seen by Critical Error Handler

flags from int 21h instruction

far return address to DOS caller

ES

DS

BP

DI

SI

DX

CX

BX

AX

flags from kernel's int 24h instruction

far return address to kernel

An application might therefore choose to pop the int 24h return address and flags from the stack, restore the registers, and return to the instruction immediately following the int 21h that encountered the error, perhaps also trying to make the result look enough like a normal failure that the application will handle the situation correctly. Indeed, programmers had little practical choice in the matter before the introduction in

DOS version 3.0 of an option to fail error-stricken int 21h calls. Even in more recent software, however, the practice is not uncommon; the Microsoft C library provides a function called `_hardretn` so that a critical error handler installed through `_harderr` may return direct to the application. There are, however, all sorts of problems with this technique; several of them stem from the possibility that the critical error may occur during the exec function.

In principle the register images on the stack are the values passed to the int 21h function. They are placed there as part of the kernel's int 21h dispatch code, before switching to an internal stack. When the DOS kernel is about to finish processing an int 21h function, it retrieves the registers from the caller's stack before issuing the IRET instruction that concludes the interrupt. To return a value to the DOS caller, the internal procedure responsible for processing a particular DOS function must deliberately enter the value into the image of the caller's saved registers on the stack; if this is not done, then all registers (including the CPU flags) will be preserved. Actually there is an exception to this principle that DOS calls preserve all registers not used to return results. When the time comes to return to the int 21h caller, the value of AL is *always* loaded onto the stack (although for some of the simpler DOS functions, this makes no difference since AL will not have changed). For those functions that indicate success or failure by the state of the carry flag, the internal procedures branch to code that adjusts the carry flag on the stack and loads a value for AX.

It should now be clear how the exec function corrupts register AX: when it calls DOS's internal procedure for opening a file, and that procedure places the *file handle* in the AX position of the saved register image on the stack, ready for return to the DOS caller. This overwrites the previously-saved value of AX, which of course was 4B00h. This is just the first time that the stacked AX value will be changed, but already the implication should be clear: a critical error handler cannot *ever* use the value in the AX image in Table 5-3 to deduce which function number has caused the problem (as does the `_hardretn` implementation in Microsoft C 6.0).

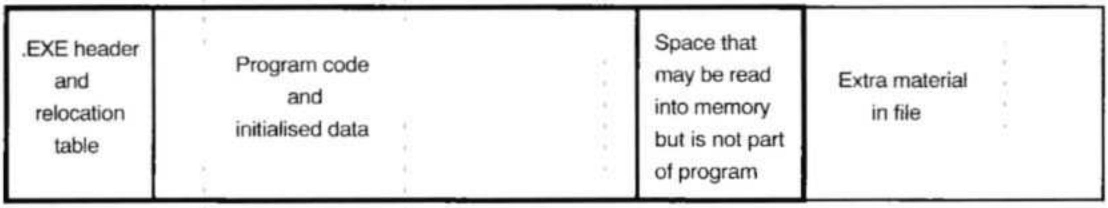
Now consider the effect on the exec function if a critical error occurs and the handler by-passes the DOS kernel in favor of returning straight to the DOS caller. It is possible that the DOS kernel has opened a file handle and obtained two memory allocations—one for the child's environment block and another for the program itself. If the critical error handler does not return to the DOS kernel, then these resources will not become available again until the current process terminates. It is arguably impossible to deal with this situation perfectly. A certain amount of cooperation is implicit in an environment that invites applications to handle operating system exceptions entirely on their own, that is, without forcing the applications to return control to the system. DOS's solution is to mark the memory blocks as free before attempting the read and then to reassign them to the current process if the read procedure returns. The exercise is shrouded in critical section signals to prevent disruption. Less can be done about the open file handle and the corresponding SFT. Since these will be needed during the read operation, it is not an option to release these resources temporarily.

The memory blocks are *marked* as free by editing the ownership field of the corresponding memory arena headers. The segment addresses for the memory blocks are kept as local variables. DOS knows that if either variable contains the value 0000h, then the corresponding block has not yet been obtained. For the sake of completeness and a little mirth, observe that if the handle obtained for the program file turns out to describe a character device, then the cleanup routines are called into action, but without yet having initialized the local variables. The result is that DOS clears two words of memory at unpredictable addresses—a small, if unlikely, bug that has persisted since at least DOS 3.20 and is testament to how easily even straightforward programming errors may go unnoticed.

Memory Layout for .EXE Programs

A program file on disk consists of an .EXE header and relocation table followed by the material to be loaded as the program's memory image. As illustrated in Figure 5-1, the file on disk may extend further than the .EXE header declares. A striking example is the WIN386.EXE file, a very large file containing a comparatively tiny real-mode .EXE program that loads the VMM and VxDs embedded in the rest of the file.

Figure 5-1: Generic Example of .EXE Program in Disk File


.EXE header and relocation table
Program code and initialised data
Space that may be read into memory but is not part of program
Extra material in file

The DOS kernel regards the program as extending from the end of the .EXE header and relocation table to the 512-byte page boundary at or beyond the end of the program material. As far as DOS is concerned, the rest of that last page is part of the program's image, whatever the programmer's intentions. This despite the presence of a field in the .EXE header (see offset 2 in Table 5-2) which gives the number of bytes used by the program in the last page. Software that loads overlays (and must therefore allow sufficient space in its own memory area) must bear this 512-byte granularity in mind. This is often described as though it were a bug in int 21h function 4B03h.

Starting with DOS 5, an initial SS:SP of 0000:0000 (as happens if no stack segment declaration is found at link time) implies a 256-byte stack appended to the program, provided the total of program plus stack will fit into 64K. This constraint ensures addressability from the program's base segment. As far as the remainder of this discussion is concerned, this stack allowance counts as part of the program's image from disk.

Having settled on a size for the program image, the function must find a sufficiently large block of free memory. For every new process, DOS creates a new Program Segment Prefix (PSP). This adds 256 bytes to the program's memory requirement. No program can ever be loaded unless there is a free block to hold the PSP and the program image.

The program itself may require space for uninitialized data such as variables whose fixed addresses are relied on by program code but which need not appear in the program's image on disk. Most high-level languages provide for this possibility. The .EXE header accommodates it through the `minalloc` field, which should be set by the linker to a value sufficiently high to cover the uninitialized data segments that are stripped from the program file. Although this uninitialized data does not contribute to the program's image on disk, it is likely to be a vital part of the program's image in memory. Thus, a memory block must be found that is large enough for the PSP, the image from disk and the `minalloc` value.

A complementary provision in the .EXE header is a `maxalloc` field, intended to cover objects handled dynamically by the program but out of its own memory—for instance, a near heap or stack. The field is interpreted as the maximum amount of memory desired in addition to the PSP and disk image.

The following pseudo-code summarizes the calculation of a .EXE program's memory requirement:

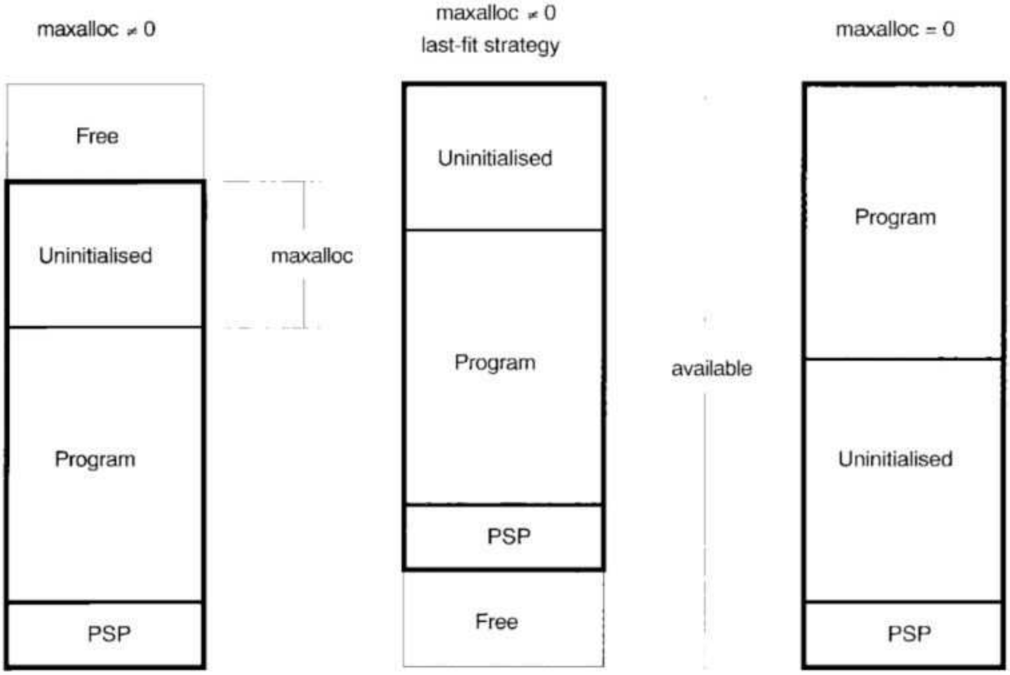
```
if (psp + program + minalloc > available) return (ERROR);
if (maxalloc == 0) allocate (available);
else allocate (min (psp + program + maxalloc, available));
```

The case where the maxalloc equals zero is further distinguished by a peculiar arrangement of program image and PSP within the allocated memory block. The program image is placed at the end of whatever memory block is found and will probably not be contiguous to the PSP.

Figure 5-2 shows the simplest level of variation depending on memory strategy and .EXE header fields. This is the situation where just the one substantial block of memory is available when int 21h function 4Bh is called. Observe that the first two cases are distinguished only if maxalloc is sufficiently small that the program is not assigned the whole block. By default, the linker sets maxalloc to FFFFh.

The memory allocation may need to be obtained in stages to support loadhigh. If the loadhigh allocation strategy indicated by bit 7 is active when int 21h function 4Bh is called, then when the time comes to find the memory for the program, the DOS kernel also sets the strategy denoted by bit 6 (the forcehigh strategy). The notion of largest available block will therefore apply only to upper memory. If the algorithm or the ensuing allocation fails, then the kernel repeats the calculations and allocation but without the strategy denoted by bit 6 (so that both conventional and upper memory count on the second attempt).

Figure 5-2: Position of Program Image Within Memory Block


maxalloc ≠ 0
Free
Uninitialised
Program
PSP
maxalloc
maxalloc ≠ 0
last-fit strategy
Uninitialised
Program
PSP
Free
available
maxalloc = 0
Program
Uninitialised
PSP

Sharp-eyed readers may have noticed an opportunity for something to go wrong. Recall from Chapter 4 that the variable in which the DOS kernel keeps the current memory allocation strategy does not lie in the SDA and is not instanced by Windows. It looks possible for a task switch to occur while the DOS kernel has changed the strategy as part of its processing of the exec function. If the incoming task has not been written with an awareness of the DOS 5 loadhigh and forcehigh strategies, it may find that all its memory allocations fail, not realizing that its requests are being constrained to upper memory only. It gets worse: If the new task tries to execute a program, then the global variables in which the DOS kernel has saved the original strategy and the memory requirement calculated for the child program being loaded by the suspended task will be overwritten, since they are not instanced either.

Relocation for .EXE Programs and Overlays

These problems aside, the memory layout for the .EXE program is essentially complete and DOS must now read the image from disk to the required position within the allocated memory block. For .EXE overlays, this position is specified by the first field in the parameter block passed to int 21h function 4Bh. The allocation of memory for overlays is the caller's (prior) responsibility.

DOS advances the file pointer past the .EXE header and starts reading the file. If there is more than 64K of program material to read, then DOS reads it in units of FE00h bytes (until less than 64K remains). A read error is fatal to the exec function. DOS tries to read the whole of the last 512-byte page that the header declares to contain program material. DOS's attempt to read this last page may produce only a partial success if the file size has been trimmed (not unreasonably) to the last byte of program material. The coding is a little slack however and DOS accepts any partial success in this situation: it is possible that some of the last page could be missing from the file and DOS would not notice.

Once the program material is in memory, DOS has to apply whatever segment fixups are indicated in the relocation table that typically follows the .EXE header. For .EXE programs, the initial SS:SP and initial CS:IP also need to be calculated, since the segments given in the .EXE header are relative to the run-time segment address of the program's base. These far pointers are stored in the kernel's data area immediately after the buffer used for the .EXE header. They are instanced by Windows.

Entries are read from the relocation table 99 at a time to a place within the Swappable Data Area. Each entry is a far pointer whose segment component must be interpreted relative to the base of the program's image in memory. After this segment adjustment, the far pointer identifies a word to which DOS must add the "relocation factor." For .EXE programs, this equals the program's load segment. For .EXE overlays, it is the second field in the parameter block passed to int 21h function 4Bh at ES:BX. DOS 5 corrects a problem with earlier versions where this relocation factor was also used when adjusting the far pointer.

As the final measure particular to the .EXE format (and only if a **dos=high** configuration has been achieved), DOS examines the executable image to see if special preparations are required. If the image starts within the first 64K of memory, DOS looks two bytes before the initial CS:IP to see if the letters "RB" are present. This is the usual test for executables packed with the EXEPACK utility that used to accompany Microsoft's language products and whose functionality is now built into the Microsoft linkers (the test is employed by SYMDEB, which raises a warning message if asked to debug a packed file). The DOS kernel is aware of three formats of EXEPACK stub code that rely on segment wrap-around at 1MB. Moreover, the kernel is equipped with corresponding run-time patches to repair the stub code before it is allowed to execute.

A second preparation is enabled only for 80286 processors when **dos=high**. In this case, DOS examines various items of the newly-loaded code, including, most notably, a version number that is given both as the very first word of the program's image and also as a decimal string elsewhere. The expected version numbers are from 3.48 to 3.83 inclusive. There cannot be much software with version numbers like this, and it has been confirmed that the DOS kernel is looking for code supplied by the Rational Systems 16-bit DOS extender, DOS/16M (used most prominently by Lotus 1-2-3 Release 3). It is far from clear just why this DOS extender needs special treatment from MS-DOS itself when **dos=high**. The patch merely replaces one sequence with a far call to code supplied by DOS which differs from the original only in that it preserves register cx (where the old code would have cleared it to zero).

Memory Layout for .COM Programs

Programs that are not in .EXE format are very much easier to handle. DOS simply obtains the largest available block of memory. Again, if the loadhigh bit is set in the allocation strategy, then the strategy is changed temporarily so that forcehigh applies as well. If a memory problem arises, then the forcehigh idea is abandoned so that the allocation can be sought from both conventional and upper memory.

An immediate memory problem would be that the largest block of free memory is too small to contain a whole PSP and leave enough space for even a tiny .COM program. DOS 5 and earlier versions fail the memory allocation stage (and thence the whole exec function unless DOS is engaged on its two-step process to load into upper memory) if there is not more than 100h bytes available. DOS 6 demands that 0200h bytes be available if the .COM file is actually to be read.

The following pseudo-code summarizes the algorithm:

```
if (allocatedbytes <= 0100h) return (ERROR);
else if (allocatedbytes <= FFFFh) read (allocatedbytes - 0200h);
else read (FF00h);
```

A read failure causes the exec function to fail. A wholly successful read causes DOS to reject the memory block as too small and therefore either to fail the exec function or to repeat the search in conventional memory. The read operation is expected to be only partially successful. Note that there is no direct attempt to check that the number of bytes read matches the file size.

The file contents are read to the memory immediately after the PSP. The first byte of the .COM file is necessarily the program's entry point, but because the initial CS is set to the PSP, the initial IP is 0100h. The .COM program will receive control with SS:SP pointing to the very last word in the allocated memory block. The kernel clears this word to zero for compatibility with programs whose notion of terminating is to issue a RET instruction (as if they are just a subroutine being called by the operating system). Offset zero with respect to CS is, of course, the start of the PSP, where there is always an int 20h instruction.

As with .EXE files, there may be some preparation peculiar to the **dos=high** configuration.

The Execution State

DOS 5 and higher contain a new EXEC subfunction, int 21h function 4B05h (Set Execution State). Few books or articles on DOS programming explain this function or its importance. The function is intended for programs that do their own EXEC, for example by intercepting int 21h function 4Bh and handling it themselves. The best example is probably the NetWare workstation shell, NETX, which provides its own EXEC so that the workstation can run executables off the file server (newer versions of NetWare provide an int 2Fh redirector rather than hook int 21h).

Table 5-4: DOS Function 4B05h—Set Execution State

INPUT:

AX	4B05h
DS:DX	address of parameter block

OFFSET	SIZE	DESCRIPTION
00h	word	unused
02h	word	bit flags
		0001h set for .EXE format; clear for .COM format
		0002h set for overlay
04h	dword	address of program/overlay pathname as ASCIIZ string
08h	word	segment address of PSP
0Ah	dword	entry point to program

OUTPUT:

cf	clear
ax	0000h
A20	disabled

The official documentation describes a dword field at offset 0Eh in the parameter block, intended to represent the program size. However, the field is unused in the function's present implementation and is therefore not shown in Table 5-4 (the function descriptions throughout this book are based on the actual implementation and not on previous documentation).

Unless bit 1 is set in the control flag, the filespec component of the ASCIIZ pathname is checked against the version table loaded by SETVER. Either the version number found in the table, or the current DOS version (such as 6.00), is copied to the relevant field of the PSP given in the function's parameter block.

The official reference designates bit 1 for overlays, suggesting that the function's version setting facility is for programs only. However, nothing in the reference or the implementation requires that the PSP belong to a new process, so it *is* possible to use the function to change the version number that will be seen by an overlay. Indeed this would be necessary for a utility that loads device drivers after system startup, since version number deception is provided when DOS function 4B03h is used to load device drivers at the normal time.

As a general principle, configuring the system with **dos=high** implies tolerance of an enabled A20 line. However, a large class of programs will not run unless the A20 line is disabled when they start executing, although they meet the general requirement by the time they call their first system function. The explanation for this curious behavior is that in many packed executables, the decompression code (which executes first, in order to unpack the "real" program) subtracts past segment 0000h and may inadvertently overwrite the HMA (i.e., the DOS kernel and important system components). A packed executable with this problem will decompress safely only if the A20 line is initially disabled. The first DOS call is taken to indicate that the program has been unpacked and would now like to be treated as any other.

If **dos=high**, function 4B05h examines the program image in memory, with the intention of *patching* (in memory) known problems in certain programs. This remedial work is carried out invisibly when programs are loaded and/or executed using DOS functions 4B00h, 4B01h and 4B03h.

As already noted, packed executables produced with older versions of Microsoft's EXEPACK program (now incorporated in the linker /EXEPACK option) subtract past segment 0000h when decompressing and may inadvertently overwrite the HMA (such as the DOS kernel and important system components) if loaded in the first 64K of memory. To guard against this, the faulty EXEPACK decompression code is patched in memory before it receives control.

As noted earlier, Versions 3.48 to 3.83 of an .EXE program (or class of programs) not known to the author require a small code patch when running on an 80286 processor (specifically). Judging from version numbers given in the book *Extending DOS*, this would appear to be the Rational Systems DOS extender, DOS/16M, which is incorporated in Lotus 1-2-3 Release 3 and higher.

A .COM program (or class of programs), again with an identity unknown to the author, must have the A20 line disabled for some time after it begins execution. It is allowed ten int 21h calls that will not automatically reenable the A20 line.

Microsoft appear to have designed int 21h function 4B05h for the benefit of Novell (it is used in NET5.COM—the Novell Netware shell for DOS 5) and neglected the internal consistency of normal DOS programming—how is a debugger or profiler to use int 21h function 4B01h to load a program and arrange for the A20 to be disabled when execution is eventually allowed to start, without having something similar to Set Execution State? In this context, the function is very badly constructed.

The following A20TEST program helps demonstrate that Microsoft's own debuggers completely miss the need to correctly set up A20 for child processes. As we will see, A20TEST's surroundings differ significantly when run under the debugger than when run standalone.

```
; A20TEST.ASM
.MODEL                       TINY
.CODE
.STARTUP
                             push     ds
                             xor      ax,ax
                             mov      ds,ax
                             mov      si,0080h  ; DS:SI = 0000:0080
                             dec      ax
                             mov      es,ax
                             mov      di,0090h  ; ES:DI = FFFF:0090
                             inc      ax
                             mov      cx,0040h
                             repz     cmpsw
                             jz       @f
                             inc      ax
@@:
                             pop      ds

                             mov      cx,ax

                             mov      dx,OFFSET header
                             mov      ah,09h
                             int      21h

                             mov      dx,OFFSET disabled
                             jcxz     @f
                             mov      dx,OFFSET enabled
@@:
                             mov      ah,09h
                             int      21h
                             mov      dx,OFFSET close
                             mov      ah,09h
                             int      21h

.EXIT                        cl            ; int 21h ah=4Ch al=cl

.CONST

header                       db       0Dh, 0Ah, "A20 status at startup:  ", "$"
disabled           db        "disabled ", "$"
enabled            db        "enabled ", "$"
```

```
close                   db      0Dh, 0Ah, "$"

END
```

By comparing the bytes at 0000:0080 and FFFF:0090, this program uses essentially the same A20 test as DOS itself: if the bytes are identical, presumably the two pointers point to the same memory location, so A20 is disabled. Running A20TEST from the DOS command line confirms that normally the A20 line is disabled when a new process is started:

```
P:\MEM\A20TEST>a20test

A20 status at startup:  disabled
```

One might expect that the version of DEBUG distributed with DOS 5 or the newer versions of CodeView would cater to this by arranging for their child process (the program being debugged) to also start with A20 disabled, but they don't:

```
P:\MEM\A20TEST>debug a20test.com
-g

A20 status at startup:  enabled

Program terminated normally
-
```

The essence of a debugger is to provide a controlled testing environment which duplicates as nearly as possible a program's normal operating conditions. DOS function 4B01h exists precisely to ensure that debuggers may load a program in exactly the same way as would happen using function 4B00h.

SPOTLOAD

The main problem for an author presenting a program loader is that there is not enough space available (nor desire, I suspect) to cover the sort of extension, PATH, and other issues that make the loader genuinely useful. The solution adopted here is to note that the natural C functional boundaries around the "load and execute" int 21h function are so close to those used by the internal `_dospawn` function called by the `spawn` family of functions in the Microsoft C library that one may as well replace `_dospawn`. This allows the loader's top-level module to use functions such as `spawnvp` exactly as usual (which means the results can be debugged with the real `_dospawn` in place, just to make sure the top level works), yet with some additional functionality, such as the ability to specify *where* a program is to be spawned. The final program has all the advantages of resolving extensions, following the PATH, and so on.

DOSPAWN.C replaces the same module in the Microsoft C library. Global variables are used to pass in extra arguments, specifying a range into which the child program is to be loaded and possibly also the memory allocation strategies which the child is to experience. Given that the selectivity was all done in MEMBLOCK.C from Chapter 4, DOSPAWN is allowed to concentrate on demonstrating the non-trivial use of int 21h function 4B01h and (under DOS 5 and higher) function 4B05h.

The SPOTLOAD program is essentially a wrapper around the `spawnvp` function, as enhanced by the new version of `_dospawn`. Somewhat like the LOADHIGH /L option in DOS 6, SPOTLOAD loads a given program into a specified memory region:

```
SPOTLOAD [/B:base [/C:ceiling]] [options] program [arguments]
```

Other SPOTLOAD options set the memory allocation environment that will apply while the child program executes:

```
/U:[OFF|ON]             enable or disable the upper memory link
/S:[FIRST|BEST|LAST]    set a bare memory allocation strategy
/L:[OFF|ON]             set a preference for upper memory (loadhigh)
/H:[OFF|ON]             set an insistence on upper memory (highonly)
/R                      restrict child to base-ceiling memory
```

SPOTLOAD gives due thought to such issues as A20 and BREAK. Unlike (as shown earlier) DEBUG and many other programs that also use int 21h function 4B01h, SPOTLOAD ensures that A20 is disabled when the child process gets control. Similarly, if BREAK is ON before SPOTLOAD runs, then it will also be ON for the child. Naturally, in DOS 5 and higher SPOTLOAD calls function 4B05h (Set Execution State). All of this is taken care of inside the `_dospawn` function, which Microsoft C uses when SPOTLOAD calls `spawnvp`.

```
/* DOSPAWN.C */

/* This file provides a generalization of DOS process management to allow
   specification of the memory region into which a program is loaded for
   execution. */

#include    <dos.h>
#include    <errno.h>
#include    <process.h>
#include    <stdlib.h>

#include    <exec.h>
```

```
#include    <fcb.h>
#include    <intpack.h>
#include    <psp.h>
#include    <standard.h>

#if NOT (defined (M_I86SM) OR defined (M_I86CM))
#error   prepared only for memory models with small data size
#endif

/*  Functions defined in MEMBLOCK.C to force memory allocations into a
    given region by masking off all other memory  */

extern WORD MaskMemoryRegion (_segment, _segment, WORD *);
extern WORD ReleaseMask (void *);

/*  Parameters defining the region of interest  */
_segment spBase = 0x0000, spCeiling = 0xFFFF;

/*  Parameters selecting a memory environment for the child program - only
    the upper memory link is configurable in this demonstration  */

WORD wChildsStrategy = 0;
BOOL bChildsUMBLink = FALSE;

BOOL bSetChildsStrategy = FALSE, bSetChildsUMBLink = FALSE;

BOOL bRestrictChildsMemory = FALSE;

/*  Functions and data to bring upper memory into the scheme  */
extern WORD _dos_getstrategy (WORD *);             // in memstrat.c
extern WORD _dos_getumblink (BOOL *);              // in memstrat.c
extern WORD _dos_setstrategy (WORD);               // in memstrat.c
extern WORD _dos_setumblink (BOOL);                // in memstrat.c

WORD wOrgStrategy = 0;
BOOL bOrgUMBLink = FALSE;

/*  Functions and data to control possible termination by Ctrl-C or
    Ctrl-Break while the upper memory link has been changed  */

extern void _dos_getbreakflag (BYTE *);            // in break.c
extern void _dos_setbreakflag (BYTE);              // in break.c
```

```
BYTE bOrgBreakFlag;
void (_interrupt _far *lpOrgInt24h) (void);

void _interrupt _far int24h (struct INTPACK r)
{
    r.al = _HARDERR_FAIL;
}

extern BOOL _child;              // for the signal function's int 23h handler

/*  Macro to ease PSP management  */
#define PSP_FP(seg) ((struct PSP _far *) MK_FP (seg, NULL))

/*  Data structures required for DOS function 4B01h  */
static struct FCB fcb1;
static struct FCB fcb2;
static struct LOADBLOCK loadblock;

/*  Data structure required for DOS function 4B05h in DOS 5  */
static struct EXECSTATE execstate;

/*  Where possible, assembly language code is placed in separate C language
    functions.      */

static WORD GetChildsExitCode (void);
static _segment GetCurrentProcess (void);
static void InitFCBs (char *);
static void SetExecState (void);

/*  Prototype for assembly language routine in Microsoft C library which
    maps DOS error numbers into the scheme presented in the ERRNO.H file. */

extern int _dosretax (void);

/*  Note that _dosretax is a common tail end for the many library functions
    that must load error values into the errno global variable. It is not
    a function in its own right but more like a dislocated piece of the
    function that jumps to it.

    Functions jump to _dosretax with a clear carry flag if they mean to
    return the value in ax as their own function value. Alternatively,
    they jump with a set carry flag if they mean to return -1 as an error
    indication (like the family of spawn functions), but need to have the
    value in ax (a DOS error) be converted and recorded.
    The C prototype given above is therefore strictly incorrect. It is
    needed however as an external reference for the following brief
    function which bridges the gap between the assembly language and the
    use of _dosretax by C functions outside the Microsoft C library. */

static int dosretax (WORD value_for_ax, BOOL value_for_cf)
{
    _asm {
            xor     ax,ax
            sub     ax,value_for_cf
            mov     ax,value_for_ax
            call    _dosretax       // _dosretax "returns" directly to
    }                                // the function that called dosretax
}

/*  The main function in this module replaces an internal component at the
    heart of all spawn functions in the Microsoft C library. */

int _dospawn (int mode, char *progname, char *cmdline, char *env)
{
    register WORD exitcode;
    BOOL convert_exitcode;
    WORD maskhandle;
    register _segment childpsp;
    BOOL execution_attempted, first_time_through;

    /*  Check the spawn mode. Under DOS, the Microsoft C library supports
        only the value P_WAIT defined in the PROCESS.H file (not counting
        overlays). If the function has passed an unsupported mode, then -1
        should be returned and the errno global variable should be loaded
        with an appropriate value from the list in ERRNO.H. */

    if (mode != P_WAIT) {
        errno = EINVAL;
        exitcode = 0xFFFF;
        convert_exitcode = FALSE;
    }
    else {

        /* Until the child program has actually been executed, all further
           values loaded into exitcode will be DOS error numbers requiring
           last minute conversion into the errno scheme. */
```

```
convert_exitcode = TRUE;

/*  Load the environment and command line addresses into the
    parameter block for the DOS "load program" function. */

loadblock.spEnvironment = (_segment) env + ((WORD) env >> 4);
loadblock.lpCommandLine = cmdline;
loadblock.lpFCB1 = &fcb1;
loadblock.lpFCB2 = &fcb2;
InitFCBs (cmdline);

if (_osmajor >= 5) {
    _dos_getstrategy (&wOrgStrategy);
    _dos_getumblink (&bOrgUMBLink);
}

/*  Bringing upper memory into play while loading the child may
    involve changing the upper memory link. If so, steps must also
    be taken to ensure that the parent program cannot terminate (via
    int 23h or 24h exceptions) until the link has been restored to
    its original condition. It is easiest just to prevent these
    exceptions from either occurring or causing any problem. Since
    the code involved will be using only those DOS functions which
    do not check Ctrl-C and Ctrl-Break unless the break flag is on,
    it suffices just to turn that flag off. Critical errors may
    occur during DOS function 4Bh (used to load the child program
    from its file on disk):  they must be intercepted and returned
    harmlessly. These complications do not arise if the upper
    memory link is already enabled. */

if (_osmajor >= 5) {
    _dos_getumblink (&bOrgUMBLink);
    if (NOT bOrgUMBLink) {
         _dos_getbreakflag (&bOrgBreakFlag);
         if (bOrgBreakFlag) _dos_setbreakflag (OFF);
         lpOrgInt24h = _dos_getvect (0x24);
         _dos_setvect (0x24, int24h);
         _dos_setumblink (ON);
    }
}

/*  To force the child program into the required memory region,
    allocate all other memory temporarily - as a sort of mask.     */
```

```
exitcode = MaskMemoryRegion (spBase, spCeiling, &maskhandle);

/*  Unless the memory masking failed, load the program. Only the
    memory between the designated base and ceiling segments will be
    available for the program's environment and main allocation.
    Use DOS function 4B01h so that execution of the child's code can
    be delayed until the memory mask has been released. This means
    the child program can be forced to load in a particular area of
    memory without affecting its subsequent memory usage (or that of
    other programs).

    Observe that when DOS function 4B01h initializes the child, it
    cannot know anything of the circumstances under which the
    child's code may eventually be executed. The parent and child
    PSPs are prepared with values that apply at the time DOS
    function 4B01h is called. This means that the instruction
    labelled termaddr in the following code will be executed not
    only as the first instruction after the child has been loaded,
    but again when the child terminates. */

if (exitcode == 0) {

    first_time_through = TRUE;
    execution_attempted = FALSE;

    _asm {
              push    ds
              pop     es
              mov     bx,offset loadblock
              mov     dx,progname
              mov     ax,4B01h
              int     21h
          termaddr:
              jc        done
              xor     ax,ax
          done:
              mov     exitcode,ax
    }
    if (execution_attempted) _child = FALSE;
}

if (exitcode != 0) {

    /*    If the child has not been loaded successfully, then only a
          little cleaning up remains to be done.In particular, if
```

```
            the upper memory link was changed, then it must be restored.
            In this case, the break flag must also be returned to its
            original condition, as must the temporary int 24h handling.
            For balance, the memory mask (if allocated) is also
            released, though of course, DOS will take care of this
            anyway when the program exits (soon). */

        if (_osmajor >= 5) {
            if (NOT bOrgUMBLink) {
                _dos_setumblink (bOrgUMBLink);
                _dos_setvect (0x24, lpOrgInt24h);
                _dos_setbreakflag (bOrgBreakFlag);
            }
        }
        ReleaseMask (&maskhandle);
    }
    else if (first_time_through) {
        first_time_through = FALSE;

        /*  The absence of an error code from DOS function 4B01h means
            that the child process has been initialized successfully.
            Moreover, it will in fact be the current process (even
            though its own code loaded from disk may not be executed
            for some time yet). */

        childpsp = GetCurrentProcess ();

        /*  A high priority must be given to ensuring that the child's
            eventual termination will be safe. The question of where
            control will resume has already been handled, but there
            remains some difficulty with restoration of this function's
            stack and registers. This must certainly be dealt with
            before the child has any opportunity to terminate.

            Ordinarily, a DOS caller's registers are saved on the
            caller's stack and the resulting SS:SP is entered into the
            DOS caller's PSP. For DOS function 4B00h, the caller
            becomes a parent;  DOS switches stacks to one provided by
            the child program, with the result that the parent's stack
            and registers are preserved until the child terminates.

            DOS function 4B01h pops the stacked registers and returns -
            just like any DOS function. The stack pointer loaded into
            the PSP remains however, so correct alignment upon the
            child's termination can be arranged by rebuilding the stack
```

```
     created when DOS function 4B01h was entered. */

_asm {
        xor     ax,ax                   // space for three words
        push    ax                      // corresponding to the
        push    ax                      // interrupt return address
        push    ax                      // and flags

        push    es                      // registers in same order
        push    ds                      // as usually saved by DOS
        push    bp                      // kernel when dispatching
        push    di                      // int 21h functions
        push    si
        push    dx
        push    cx
        push    bx
        push    ax                      // this brings the stack down
}                                       // to the restoration address
                                        // recorded in the parent PSP

/*   The same clean up as before is required, but there is an
     additional subtlety regarding the temporary int 24h
     handling, for the address of this function's temporary
     int 24h handler will also have been recorded in the child
     PSP. Note that break-checking may not be reenabled until
     the int 24h address in the PSP has been restored. */

if (_osmajor >= 5) {
     if (NOT bOrgUMBLink) {
          _dos_setumblink (bOrgUMBLink);
          _dos_setvect (0x24, lpOrgInt24h);
          PSP_FP (childpsp) -> lpInt24h = lpOrgInt24h;
          _dos_setbreakflag (bOrgBreakFlag);
     }
}

/*   By default, the child process is allowed access to memory
     which had been masked off during the loading stage - thus,
     the memory mask is released before the child's real code
     gets to run. As a testing option however, the opportunity
     is given to restrict the child's execution to the loading
     area, as well. */

if (NOT bRestrictChildsMemory) ReleaseMask (&maskhandle);
if (bSetChildsStrategy) _dos_setstrategy (wChildsStrategy);
if (bSetChildsUMBLink) _dos_setumblink (bChildsUMBLink);

/*    If DOS is using the HMA, then the A20 line should be
      disabled when the child program starts executing. DOS
      function 4B05h must be the last DOS function used before
      control is transferred. Refer to the main text for a
      discussion on how this step has clearly not been planned
      carefully (if at all) in the design on DOS 5. */

if (_osmajor >= 5) {
      execstate.wFlags = EXEC_OVERLAY;
      execstate.lpName = progname;
      execstate.spPSP = childpsp;
      execstate.lpEntryPoint = loadblock.lpEntryPoint;
      SetExecState ();
}

/*    Until now, a Ctrl-C or Ctrl-Break would have terminated
      the child process before any of its code (loaded from
      its file on disk) had a chance to execute.      Mark
      the barrier with a Boolean variable so that bogus
      termination can be detected. */

execution_attempted = TRUE;

_child = TRUE;

/*    Load the child program's stack. A value to be passed in
      register AX must be popped off. Place the child's entry
      point onto the stack, ready for transfer via a RETF
      instruction. Make sure that DS and ES both contain the
      segment address of the child's PSP. */

_asm {
          mov     dx,childpsp
          mov     ss, word ptr loadblock.lpStackTop+2
          mov     sp, word ptr loadblock.lpStackTop
          pop     ax
          push    word ptr loadblock.lpEntryPoint+2
          push    word ptr loadblock.lpEntryPoint
          mov     es,dx
          mov     ds,dx
          retf
}                    // control resumes at the label termaddr (the
            goto termaddr;// (goto statement is merely a reminder), after
        }                         // which consultation of a Boolean variable
        else {                    // directs control to the following block

            if (bSetChildsStrategy) _dos_setstrategy (wOrgStrategy);
            if (bSetChildsUMBLink) _dos_setumblink (bOrgUMBLink);
            if (bRestrictChildsMemory) ReleaseMask (&maskhandle);

            /*  Retrieve the child's exit code:  after successful execution
                and termination of the child program, this is the value
                expected to be returned by the spawn functions.

                Beware however, of bogus termination caused by exceptions
                to DOS functions used between the initialization of the
                child as a process (i.e., its loading) and the execution of
                the child's real program code. In this case, there is no
                exit code from the child and the spawn functions should
                report that they were unable (for presumably temporary
                reasons) to start the child. */

            if (execution_attempted) {
                exitcode = GetChildsExitCode ();
            }
            else {
                errno = EAGAIN;
                exitcode = 0xFFFF;
            }
            convert_exitcode = FALSE;     // these exit codes are final
        }
    }
    return (dosretax (exitcode, convert_exitcode));
}

void InitFCBs (char *cmdline)
{
    _asm {                                      // compiler preserves si, di
            mov     si,cmdline
            inc     si                          // advance past length count
                                                // that starts command line
            push    ds
            pop     es
            mov     di,offset fcb1
            mov     ax,2901h
            int     21h                 // DOS parse FCB function advances
                                        // si past argument (if present),
                mov     di,offset fcb2  // ready for next
                mov     ax,2901h
                int     21h
    }
}

static _segment GetCurrentProcess (void)
{
    _asm {
                mov     ah,51h
                int     21h
                mov     ax,bx
    }
}

static void SetExecState (void)
{
    _asm {
                mov     dx,offset execstate
                mov     ax,4B05h
                int     21h
    }
}

static WORD GetChildsExitCode (void)
{
    _asm {
                mov     ah,4Dh
                int     21h
    }
}
```

SPOTLOAD.C is the top-level module for the SPOTLOAD program, concerned almost entirely with parsing its fairly complicated command line, then calling **spawnvp**, with extra parameters in global variables. In turn, **spawnvp** will indirectly call our new **_dospawn**.

```
/* SPOTLOAD.C - top-level module for SPOTLOAD.EXE */

#include    <dos.h>
#include    <process.h>
#include    <stdarg.h>
#include    <stdio.h>
#include    <stdlib.h>
#include    <string.h>
```

```
#include    <arena.h>
#include    <standard.h>

char description [] = "\n\
Loads a given program into a specified memory region\n\
\n\
  SPOTLOAD [/B:base [/C:ceiling]] [options] program [arguments]\n\
\n\
Base and ceiling segment addresses are expected in hexadecimal form.\n\
Use /R to restrict the child program to this same area of memory.\n\
\n\
Other options set the memory allocation environment that will apply\n\
while the child program executes:-\n\
\n\
  /U:[OFF|ON]               enable or disable the upper memory link\n\
  /S:[FIRST|BEST|LAST]      set a bare memory allocation strategy\n\
  /L:[OFF|ON]               set a preference for upper memory (loadhigh)\n\
  /H:[OFF|ON]               set an insistence on upper memory (highonly)\n\
";

/*  Variables to configure the child program's memory operations during its
    loading and executing stages  */

extern _segment spBase, spCeiling;                  // while loading

extern WORD wChildsStrategy;                        // remainder apply
extern BOOL bChildsUMBLink;                         // while child executes
extern BOOL bSetChildsStrategy, bSetChildsUMBLink;

extern BOOL bRestrictChildsMemory;

/*  Functions for strategy control  */
extern WORD _dos_getstrategy (WORD *);              // in memstrat.c
extern WORD _dos_getumblink (BOOL *);               // in memstrat.c
extern WORD _dos_setstrategy (WORD);                // in memstrat.c
extern WORD _dos_setumblink (BOOL);                 // in memstrat.c

static BOOL ReadStrategy (char *, WORD *);
static char *strategies [] = {"first", "best", "last", NULL};

/*  Miscellany      */
extern BOOL ReadBoolean (char *, BOOL *);
extern BOOL ReadHexWord (char *, WORD *);
```

```
void PutError (char *, ...);

/*  ==================================================================  */

int main (int argc, char **argv)
{
    register CHAR *ptr;
    register WORD ch;
    WORD temp;
    int exitcode;

    /*  Reject early DOS versions. */
    if (MK_WORD (_osmajor, _osminor) < MK_WORD (3, 10)) {
        PutError ("requires DOS version 3.10 or higher");
        return (-1);
    }

    /*  Obtain (run-time) default values for the child program's memory
        environment. */

    if (_osmajor >= 5) {
        _dos_getstrategy (&wChildsStrategy);
        _dos_getumblink (&bChildsUMBLink);
    }

    /*  Process the command line. */
    while (++ argv, -- argc) {
        ptr = *argv;
        if (*ptr == '/') {
            ch = ptr [1] | (BYTE) 0x20;
            if (ptr [2] == '\0') {
                if (ch == '?') {
                    printf (description);
                    return (0);
                }
                else if (ch == 'r') bRestrictChildsMemory = TRUE;
            }
            else if (ptr [2] == ':') {
                if (ch == 'b' OR ch == 'c') {
                    if (NOT ReadHexWord (ptr + 3, &temp)) {
                        PutError ("bad number in switch %s", ptr);
                        return (-1);
                    }
                    if (ch == 'b') spBase = temp;
                    else spCeiling = temp;                 // if (ch == 'c')
    }
    else if (ch == 'h' OR ch == 'l' OR ch == 'u') {
        if (NOT ReadBoolean (ptr + 3, &temp)) {
            PutError ("bad Boolean in switch %s", ptr);
            return (-1);
        }
        if (_osmajor >= 5) {
            if (ch == 'h') {
                if (temp) temp = HIGHONLY;
                temp = (wChildsStrategy & HIGHONLY) ^ temp;
                if (temp) {
                    wChildsStrategy ^= temp;
                    bSetChildsStrategy = TRUE;
                }
            }
            else if (ch == 'l') {
                if (temp) temp = LOADHIGH;
                temp = (wChildsStrategy & LOADHIGH) ^ temp;
                if (temp) {
                    wChildsStrategy ^= temp;
                    bSetChildsStrategy = TRUE;
                }
            }
            else {                 // if (ch == 'u')
                if (bChildsUMBLink ^ temp) {
                    bChildsUMBLink = temp;
                    bSetChildsUMBLink = TRUE;
                }
            }
        }
    }
    else if (ch == 's') {
        if (NOT ReadStrategy (ptr + 3, &temp)) {
            PutError ("bad strategy in switch %s", ptr);
            return (-1);
        }
        if (_osmajor >= 5) {
            temp = (wChildsStrategy & STRATEGY_MASK) ^ temp;
            if (temp) {
                wChildsStrategy ^= temp;
                bSetChildsStrategy = TRUE;
            }
        }
    }
}
```

```
              else {
                    PutError ("invalid switch %s", ptr);
                    return (-1);
              }
          }
          else break;
    }
    if (argc == 0) {
          PutError ("requires name of program to load");
          return (-1);
    }

    exitcode = spawnvp (P_WAIT, *argv, argv);

    if (exitcode == -1) PutError ("%s", strerror (errno));
    return (exitcode);
}

static BOOL ReadStrategy (char *ptr, WORD *result)
{
    register char **mark;
    char *str;
    WORD temp;

    mark = strategies;
    temp = FIRST_FIT;
    while (str = *mark ++, str != NULL) {
          if (stricmp (str, ptr) == 0) {
              *result = temp;
              return (TRUE);
          }
          temp ++;            // successive strategies differ by 1
    }
    return (FALSE);
}

void PutError (char *str, ...)
{
    va_list va_start (argptr, str);
    printf ("\nSPOTLOAD fatal error:  ");
    vprintf (str, argptr);
    printf ("\n");
    va_end (argptr);
}
```

READBOOL.C is fairly obvious (it reads boolean strings such as "true", "on", and "yes") and is separated out like READHEX.C earlier, even though it is not used in any of this book's other programs.

```
/* READBOOL.C */

#include    <string.h>

#include    <standard.h>

static char *strings [] = {"false", "off", "no", "0", NULL,
                           "true", "on", "yes", "1", NULL};

BOOL ReadBoolean (char *ptr, BOOL *result)
{
    register char **mark;
    char *str;
    register BOOL temp;

    mark = strings;
    temp = FALSE;
    for (;;) {
          while (str = *mark ++, str != NULL) {
               if (stricmp (str, ptr) == 0) {
                     *result = temp;
                     return (TRUE);
               }
          }
          if (temp) return (FALSE);
          temp = TRUE;
    }
}
```

Recall that, in an earlier test, running DEBUG A20TEST.COM demonstrated that many debuggers fail to disable A20 for new child processes. In contrast, SPOTLOAD gives the correct result:

```
P:\MEMMAN\SPOTLOAD>spotload ..\a20test\a20test

A20 status at startup:  disabled
```

Because Microsoft's **spawn** functions allow command-line execution of programs with extensions other than .COM and .EXE, a possibly beneficial side ffect of using

SPOTLOAD, even when the precision loading is not required, would be the ability to run a program with a non-standard extension, such as the CPWIN386.CPL Control Panel "applet" from Windows 3.1.

Changing FILES= After CONFIG.SYS

The ADDFILES program demonstrates how an understanding of process termination can be used to solve an age-old problem: enlarging FILES= without rebooting. ADDFILES adds a specified number of System File Table (SFT) entries. Under DOS 5 and higher, ADDFILES first seeks upper memory for space for the new SFTs. If this fails, by default it continues searching in conventional memory. The ADDFILES /u switch insists on trying upper memory only.

ADDFILES.C is the top-level module. There is, in fact, not much else to the program, except for SFT specifics, which are given in a separate file called SFT.C (not surprisingly).

ADDFILES reloads itself with a last-fit memory-allocation strategy into higher memory (similar to SPOTLOAD) to make way for the new SFTs. This creates two "instances" of ADDFILES. The child instance makes its termination address go to the grandparent process, rather than to its parent (i.e., the first instance of ADDFILES), then it kills the parent by running its copy of the C `exit` function (see the `KillParent` function in the following ADDFILES.C).

```
/* ADDFILES.C - primary module for addfiles.exe */

#include    <dos.h>
#include    <memory.h>
#include    <process.h>
#include    <signal.h>
#include    <stdarg.h>
#include    <stdio.h>
#include    <stdlib.h>
#include    <string.h>

#include    <arena.h>
#include    <psp.h>
#include    <sft.h>
#include    <standard.h>

#pragma intrinsic (_disable, _enable)

static char description [] = "\n\
Adds a specified number of System File Tables (SFTs).\n\
\n\
  ADDFILES [ExtraSFTs] [/u]\n\
\n\
If the number of new SFTs is not given, ADDFILES reports the current count.\n\
\n\
Under DOS 5 and higher, space for the new SFTs will be sought in upper\n\
memory first. If this fails, the search continue in conventional memory\n\
by default. Use the /u switch to insist on trying upper memory only.\n";

/*  Functions and data to control memory allocation  */
extern WORD _dos_getstrategy (WORD *);              // in memstrat.c
extern WORD _dos_getumblink (BOOL *);               // in memstrat.c
extern WORD _dos_setstrategy (WORD);                // in memstrat.c
extern WORD _dos_setumblink (BOOL);                 // in memstrat.c

WORD wOrgStrategy;
BOOL bOrgUMBLink;
BOOL bDemandUpper = FALSE;       // set to TRUE if /u switch present

/*  Functions and data for SFT location and construction  */
extern struct SFT_HEADER _far *GetFirstSFTHeader (void);      // in sft.c
extern WORD GetSFTCount (void);                               // in sft.c
extern WORD GetSFTSize (void);                                // in sft.c

void BuildNewSFTs (_segment, WORD, WORD);
void AddSFTBlock (struct SFT_HEADER _far *, WORD);

WORD wAllocParas, wNewSFTs = 0;

/*  Functions and data concerned with reloading program into higher memory
    to make way for new SFTs  */

BOOL bReloaded;
_segment spParentCode, spParentData, spParentPSP;

BOOL HaveBeenReloaded (void);
void KillParent (void);

/*  Exit handling  */
int _far HardErrorTrap (void);
void cleanup (void);
void PutError (char *, ...);

/*  ======================================================================  */

int main (int argc, char **argv)
{
    char *progname;
    register char *ptr;
    BYTE ch;
    WORD sftcount, sftsize, sftblockbytes;
    register WORD errorcode;
    WORD allocseg;

    if (MK_WORD (_osmajor, _osminor) < MK_WORD (3, 10)) {
          PutError ("requires DOS version 3.10 or higher");
          return (-1);
    }

    /*    It simplifies matters greatly if premature termination via the
          int 23h and int 24h DOS exceptions can be discounted. */

    signal (SIGINT, SIG_IGN);
    _harderr (HardErrorTrap);

    /*    The program may be reloaded with a LAST_FIT strategy in order to
          move it out of the way. If this is the first instance, then parse the
          command line and determine whether or not the new SFTs can be added
          immediately. If this is the reload, perform the necessary memory
          juggling and then build the new SFTs. */

    bReloaded = HaveBeenReloaded ();
    if (NOT bReloaded) {

          /*  Process the command line, but save the program name for later
              in case reloading is required. */

          progname = *argv;    // program name in argv[0]
          while (++ argv, -- argc) {
              ptr = *argv;
              if (*ptr == '/') {
                    ch = ptr [1] | (BYTE) 0x20;
                    if (ptr [2] == '\0') {
                        if (ch == '?') {
                              printf (description);
                              return (0);
```

```
                }
                if (ch == 'u') {
                        bDemandUpper = TRUE;
                        continue;
                }
            }
            PutError ("invalid switch %s", ptr);
            return (-1);
        }
        else {
            if (wNewSFTs != 0) {
                PutError ("too many parameters");
                return (-1);
            }
            wNewSFTs = atoi (ptr);
            if (wNewSFTs == 0) {
                PutError ("invalid parameter %s", ptr);
                return (-1);
            }
        }
    }

    /*  The current SFT count is required anyway - in case the proposed
        addition would pass the fixed limit of 255 - so use it for a
        little report if no command line parameters were detected.     */

    sftcount = GetSFTCount ();
    if (wNewSFTs == 0) {
        printf ("\nThere are presently %d SFTs\n", sftcount);
        return (0);
    }
    if (sftcount + wNewSFTs > 0xFF) {
        PutError ("total would exceed 255 - can add only %d new SFTs",
                        0xFF - sftcount);
        return (-1);
    }

    /*  Calculate the size of the external memory allocation in which
        the new SFTs will be built. Include a paragraph for a memory
        arena subheader under DOS 4.0 and higher. */

    sftsize = GetSFTSize ();
    if (sftsize == 0) {
        PutError ("cannot determine size of System File Table");
        return (-1);
```

```
}
sftblockbytes = sizeof (struct SFT_HEADER) + sftsize * wNewSFTs;
wAllocParas = (sftblockbytes + 15) >> 4;
if (_osmajor >= 4) wAllocParas ++;

/*  Now that the preliminaries are over, obtain the current memory
    allocation strategy (and UMB link in DOS 5 and higher), then
    install a C run-time terminator to ensure restoration when the
    program exits. */

_dos_getstrategy (&wOrgStrategy);
if (_osmajor >= 5) _dos_getumblink (&bOrgUMBLink);

atexit (cleanup);

/*  In DOS 5 and higher, attempt first to find upper memory for the
    new SFTs. This is the easiest case. There is an assumption
    here that the program is being run from conventional memory, so
    the danger of fragmentation does not arise - in the present
    implementation therefore, the program should not itself be
    loaded high. */

if (_osmajor >= 5) {
    _dos_setstrategy (wOrgStrategy | HIGHONLY);
    errorcode = _dos_setumblink (ON);
    if (errorcode == 0) {
         errorcode = _dos_allocmem (wAllocParas, &allocseg);
         if (errorcode == 0) {
             /* BuildNewSFTs is expected to arrange the
                  retention of the memory used for the new SFTs - a
                  natural consequence of emulating the usual
                  construction of SFTs in the SYSINIT configuration
                  block. The program itself does not stay resident. */

             BuildNewSFTs (allocseg, wAllocParas, wNewSFTs);
             return (wNewSFTs);
         }
    }
    if (bDemandUpper) {
         PutError ("insufficient free upper memory");
         return (-1);
    }
}
```

```
        /*  An external allocation for the new SFTs would probably result in
            memory fragmentation. Neither is it particularly desirable to
            retain the 256 bytes of the PSP - which would happen if the
            program finished with the TSR exit, having built the new SFTs
            over space presently occupied by the program's code. Instead,
            the aim is to reload the program elsewhere and have the second
            copy (higher in memory) terminate the first, thereby securing
            its memory and avoiding fragmentation. */

        _dos_setstrategy (LAST_FIT);
        if (_osmajor >= 5) _dos_setumblink (OFF);

        errorcode = spawnl (P_WAIT, progname, progname, NULL);

        PutError ("unable to reload into higher memory (%s)",
            errorcode == -1 ? strerror (errno) : "internal error");
        return (-1);
    }
    else {
        /*  This block executes because the program has been reloaded. Two
            parameters describing the required allocation must be retrieved
            from the first instance of the program. */

        // note similarity to GetInstanceData in Windows
        wAllocParas = *((WORD _far *) MK_FP (spParentData, &wAllocParas));
        wNewSFTs = *((WORD _far *) MK_FP (spParentData, &wNewSFTs));

        /*  Take over from the first instance of the program, then allocate
            memory and build the new SFTs just as if the reload never
            happened. The parent's termination, incidentally, causes the
            memory allocation strategy to be restored, so there is no need
            for special exit handling in this reload case. */

        KillParent ();
        errorcode = _dos_allocmem (wAllocParas, &allocseg);
        if (errorcode != 0) {
            PutError ("error allocating memory after relocation");
            return (-1);
        }
        BuildNewSFTs (allocseg, wAllocParas, wNewSFTs);
        return (wNewSFTs);
    }
}

void BuildNewSFTs (_segment allocseg, WORD paras, WORD wNewSFTs)
```

```
{
    _segment arenaseg, sftblockseg, subarenaseg;

    /*    Clear the memory block. */
    _fmemset (MK_FP (allocseg, NULL), 0, paras << 4);

    #define arenaptr(seg) ((struct ARENA_HEADER _far *) MK_FP (seg, NULL))
    #define sftheaderptr(seg) ((struct SFT_HEADER _far *) MK_FP (seg, NULL))

    /*    If the DOS version is 4.0 or higher, the first paragraph of the
          allocation is intended as a memory arena subheader - to emulate the
          way that SYSINIT builds its SFT block.The MEM program uses this
          construction to identify the new SFTs as an extra FILES= entry. */

    if (_osmajor >= 4) {
          sftblockseg = allocseg + 1;
          subarenaseg = allocseg;
          arenaptr (subarenaseg) -> cSignature = 'F';
          arenaptr (subarenaseg) -> spOwner = sftblockseg;
          arenaptr (subarenaseg) -> wParagraphs = paras - 1;
    }
    else sftblockseg = allocseg;

    /*    Append the new SFT block to the existing chain. */
    AddSFTBlock (MK_FP (sftblockseg, NULL), wNewSFTs);

    /*    Assign the memory arena header as if it had been obtained for
          configuration during system startup. This reassignment also serves
          as the way to keep the block after the program terminates. */

    arenaseg = allocseg - 1;
    arenaptr (arenaseg) -> cName [0] = 'S';
    arenaptr (arenaseg) -> cName [1] = 'D';
    arenaptr (arenaseg) -> spOwner = ARENA_DOS;
}

void AddSFTBlock (struct SFT_HEADER _far *newsftheader, WORD wNewSFTs)
{
    struct SFT_HEADER _far *ptr;

    /*    Initialize the header for the new SFTs. */
    newsftheader -> lpNext = (struct SFT_HEADER _far *) -1;
    newsftheader -> wCount = wNewSFTs;

    /*   Follow the chain of SFT headers to its end, then patch in the
         address of the new SFTs. Remember to disable interrupts while
         loading the far pointer. */

    ptr = GetFirstSFTHeader ();
    while (*((WORD _far *) ptr) != -1) {
         ptr = ptr -> lpNext;
    }
    _disable ();
    ptr -> lpNext = newsftheader;
    _enable ();
}

int _far HardErrorTrap (void)
{
    return (_HARDERR_FAIL);
}

void cleanup (void)
{
    _dos_setstrategy (wOrgStrategy);
    if (_osmajor >= 5) _dos_setumblink (bOrgUMBLink);
}

void PutError (char *str, ...)
{
    va_list va_start (argptr, str);
    printf ("\nADDFILES fatal error:  ");
    vprintf (str, argptr);
    printf ("\n");
    va_end (argptr);
}

/*  ==================================================================  */

/*  The next function is concerned with identifying whether this program is
    being rerun by an earlier copy of itself in order to move out of the
    way for the new SFTs. */

static char idstring [] = "ADDFILES";

BOOL HaveBeenReloaded (void)
```

```
{
    #define pspptr(seg) ((struct PSP _far *) MK_FP (seg, NULL))

    /*  Inspect this program's PSP to find the parent's PSP. Work on the
        assumption that they are just different instances of the same
        program. Record the parent's presumed code and data segments in
        global variables for later use, then check that the ID string is at
        the same offset in both the parent's and this program's DGROUP. */

    spParentPSP = pspptr (_psp) -> spParentPSP;
    spParentCode = spParentPSP + _segname ("_CODE") - _psp;
    spParentData = spParentPSP + _segname ("_DATA") - _psp;
    if (_fmemcmp (MK_FP (spParentData, &idstring), &idstring,
            sizeof (idstring)) == 0) return (TRUE);
    else return (FALSE);
}

/*  Some highly compiler-dependent unpleasantries - explained below  */
typedef void _interrupt _far INTERRUPT_HANDLER (void);

extern INTERRUPT_HANDLER _far *_aintdiv;
extern INTERRUPT_HANDLER _cintDIV;

void KillParent (void)
{
    void _far *stackmarker;
    void (_far *retaddr) (void);

    /*  Acquire the parent's termination "state" - this program will be
        going back to its parent's parent. */

    #if FALSE
    pspptr (_psp) -> lpInt22h = pspptr (spParentPSP) -> lpInt22h;
    pspptr (_psp) -> lpInt23h = pspptr (spParentPSP) -> lpInt23h;
    pspptr (_psp) -> lpInt24h = pspptr (spParentPSP) -> lpInt24h;
    pspptr (_psp) -> spParentPSP = pspptr (spParentPSP) -> spParentPSP;
    #else
    struct TERM_STATE {                           // very efficient method of
        INTERRUPT_HANDLER _far *lpInt22h;         // copying consecutive
        INTERRUPT_HANDLER _far *lpInt23h;         // fields in structure -
        INTERRUPT_HANDLER _far *lpInt24h;         // compiler can't possibly
        _segment spParentPSP;                     // miss the optimization
    };
    #define termstateptr(seg)         ((struct TERM_STATE _far *) \
            &(((struct PSP _far *) ((DWORD) (seg) << 16)) -> lpInt22h))
```

```
*termstateptr (_psp) = *termstateptr (spParentPSP);
#endif

/*   Skipping a generation invalidates the int 00h vector recorded by the
     C run-time. The compiler dependence introduced to overcome this is
     best viewed as just desserts for interfering with another process
     while having only incomplete control of one's own program. As an
     alternative to using the run-time libraries particular labels,
     consider trapping the parent's restoration of the int 00h vector. */

#define lpParents_aintdiv           ((INTERRUPT_HANDLER _far * _far *) \
                                    MK_FP (spParentData, &_aintdiv))
_aintdiv = *lpParents_aintdiv;
*lpParents_aintdiv = _cintDIV;

/*   Set a new state for the parent - to return further down this
     procedure when it terminates. */

_asm {                                              // in-line assembler
        mov     word ptr retaddr,offset resume      // needed to resolve
        mov     word ptr retaddr+2,cs               // forward reference
}                                                   // to code label
pspptr (spParentPSP) -> lpInt22h = retaddr;
pspptr (spParentPSP) -> lpInt23h = _dos_getvect (0x23);
pspptr (spParentPSP) -> lpInt24h = _dos_getvect (0x24);
pspptr (spParentPSP) -> spParentPSP = _psp;

/*   Now that the parent-child relationship has been reversed, the stack
     adopted when the parent terminates will be taken from the relevant
     field in this program's PSP. Build a stack with registers in the
     order expected by the DOS kernel and load the address into the PSP. */

_asm {
        xor     ax,ax
        push    ax                  // space for flags and return address -
        push    ax
        push    ax

        push    es                  // registers to be recovered at end of
        push    ds                  // function
        push    bp
        push    di
        push    si
        push    dx
        push    cx
```

```
            push    bx
            push    ax

            mov     word ptr stackmarker,sp
            mov     word ptr stackmarker+2,ss
    }
    pspptr (_psp) -> lpStack = stackmarker;

    /*   Make the parent the active process and terminate it. Ordinarily,
         this would mean calling int 21h function 4Ch, but such preemption
         is unnecessary here because the parent is known;  indeed, it is
         another copy of this same program, so it is possible to call the
         exit (0) function in the parent's code segment and allow it the
         chance to clean up properly. */

/* Of course, only know the offset of exit in parent process because it's just
an earlier instance of this process. */

    _asm {
            mov     bx,spParentPSP
            mov     ah,50h
            int     21h

            xor     ax,ax                // argument on stack for exit ()
            push    ax
            mov     ax,offset exit       // place segment and offset of
            push    spParentCode         // parent's exit function on stack
            push    ax                   // ready for transfer via RETF

            mov     ax,spParentData      // load ds with parent's DGROUP
            mov     ds,ax

            retf                         // go to parent - the earlier
        resume:                          // manipulations bring control
    }                                    // back here
}
```

The `GetSFTSize` function in SFT.C uses what seems like an obvious technique to determine the size of an SFT entry. It calls the DOS internal Get SFT Entry function (int 2Fh function 1216h) to get the addresses of successive SFTs, from which the spacing from one SFT to the next may be determined. Perhaps this method only seems obvious, because as several parts of Windows use an entirely different (and extremely risky) method to determine the size of an SFT entry: opening up CON five times, searching through the first 512K of memory for the string "CON", and then subtracting its location from the next occurrence. Placing the string "CON CON CON" in a normal DOS device driver can thus fool Windows into thinking that each SFT entry is only four bytes wide!

```
/* SFT.C - a small collection of SFT management routines */

#include     <sft.h>
#include     <standard.h>
#include     <sysvars.h>

static struct SFT _far *GetIndexedSFT (WORD);

struct SFT_HEADER _far *GetFirstSFTHeader (void)
{
    struct SYSVARS _far *sysvarsptr;

    // straightforward: just life the required address
    // from the SYSVARS structure in the DOS kernel
    _asm {
             mov     ah,52h
             int     21h
             mov     word ptr sysvarsptr,bx
             mov     word ptr sysvarsptr+2,es
    }
    return (sysvarsptr -> lpSFTs);
}

WORD GetSFTCount (void)
{
    struct SFT_HEADER _far *sftheader;
    WORD count;

    /*   Follow the chain of SFT blocks, noting the number of SFTs in each
         and accumulating a total. */

    sftheader = GetFirstSFTHeader ();
    count = 0;
    do {
         count += sftheader -> wCount;
         sftheader = sftheader -> lpNext;
    } while ((WORD) sftheader != -1);
    return (count);
}

WORD GetSFTSize (void)
{
    struct SFT _far *sftptr, _far *sftmark = NULL;
    register WORD index, sftsize = 0;
    WORD count = GetSFTCount ();
    WORD diff;

    for (index = 0; index < count; index ++) {

        /*  Get the address of the next SFT. Since the index is limited by
            the count obtained before entering the loop, failure to obtain
            an SFT address would be an exception situation that warrants
            failing the function. */

        sftptr = GetIndexedSFT (index);
        if (sftptr == NULL) return (0);

        /*  If this SFT and the one before it both lie in the same block,
            then the difference of their addresses should be the SFT size.
            Record this if it is not yet known, else verify consistency. */

        if ((_segment) sftptr == (_segment) sftmark) {
            diff = (BYTE _far *) sftptr - (BYTE _far *) sftmark;
            if (sftsize == 0) sftsize = diff;
            else if (sftsize != diff) return (0);
        }

        sftmark = sftptr;
    }
    return (sftsize);
}

struct SFT _far *GetIndexedSFT (WORD index)
{
    _asm {                            // compiler preserves register di
            mov     bx,index
            mov     ax,1216h          // undocumented function takes index
            int     2Fh               // in BX and returns with es:di
            jc        fail            // addressing the corresponding SFT

            mov     ax,di             // carry flag clear on success
            mov     dx,es
            jmp     done

        fail:                         // carry flag set to indicate error
            xor     ax,ax
            cwd
        done:
    }
}
```

Unlike most utilities for changing FILES= after SYSINIT, the new SFT entries added by ADDFILES even show up correctly as "FILES=" in a report from the DOS MEM utility; see the `BuildNewSFTs` function in ADDFILES.C to see how this is done. For example, after `addfiles 10` added a new SFT with ten entries at DC98:0000, `mem /p` reported:

```
 0DC960      IO           000270    System Data
                          000260     FILES=
```

While the SFT itself was located at 0DC980h, MEM shows 0DC960h because it is including the MCB and the suballocation header.

PART 2

Some Development Tools

To study system organization it is vital to write small tools that stay resident or load as device drivers. Without these, many readers would be left to ponder system configuration as somebody else's academic discourse.

A Sample—Demonstrating the Memory Fudge for Devicehigh

Consider the claim made in Part 1 that when SYSINIT loads a device driver into a UMB, it revises the word at 0040:0013 to cover all memory up to the end of that UMB. How may this behavior be demonstrated to sceptical readers? It is one thing to write a program that inspects the word at 0040:0013 to determine an upper bound on conventional memory:

```
/* BIOSMEM.C */
#include    <stdio.h>

int main (void)
{
    printf ("\nCeiling segment for conventional memory is %04Xh\n",
          *((unsigned _far *) 0x00400013) << 6);
    return (0);
}
```

But this does not help establish that the value at this address is altered when device drivers are loaded in response to a **devicehigh** directive. Establishing this claim demands not a program which executes in the normal fashion, but a device driver whose initialization command performs the same task as the preceding program and then fails to install. Moreover, this objective should easily achieved; after all, this is just a gadget to demonstrate a small curiosity. It is very important, though, to the small class of device drivers that would otherwise be broken if loaded into upper memory.

The material presented in the next two chapters allows easy development of gadgets to probe the system during its configuration. An existing program, such as the preceding, written for and tested under normal conditions, may be adapted to load as a device driver simply by relinking with extra components.

DRIVER.ASM is a small module that provides the low-level structure expected of device drivers, along with public hooks on which other modules (written in some high-level language) may hang their implementations of the device driver commands. This is hardly special. It is the minimum that readers may expect from a treatise on writing device drivers in C.

The second component, CRTDRVR.LIB, supplies DRIVER.ASM with a device driver initialization command that reproduces the normal entry conditions of a DOS executable (save only that the entry point is assumed, for this book, to have the `astart` label used in Microsoft C). This is the key to transparent adaptation; for the C programmer (possibly after adaptation for another compiler), the `main` function serves equally well as the high-level entrance either to a normal program or to device-driver initialization.

For a simple illustration of these modules at work, take the source file given previously. Call it BIOSMEM.C and compile it exactly as before, but relink with the device driver components:

```
link driver+crtdrvr.lib+biosmem,biosmem;
```

The resulting .EXE file can still be executed normally and can be loaded as a device driver. This makes it a trivial matter to demonstrate that SYSINIT adjusts the BIOS memory record when loading device drivers into upper memory. Simply reboot with a CONFIG.SYS file containing lines such as

```
device c:\dos\himem.sys
device c:\windows\emm386.exe noems ram=c800-efff
device e:\temp\biosmem.exe
devicehigh e:\temp\biosmem.exe
dos umb
```

to see BIOSMEM report differently, depending on whether it is loaded through **device** or **devicehigh**:

```
Ceiling segment for conventional memory is A000h

Ceiling segment for conventional memory is F000h
```

By default the DRIVER module is assembled with a logical device name consisting only of spaces; unless the driver is staying in memory to interact with the DOS kernel through the established device driver interface, the logical device name is probably superfluous. The DRIVER module can be reassembled with a logical device name by defining the symbol `_NAME_` on the assembler command line. For example, the special treatment that SYSINIT accords a device driver with logical name "PROTMAN$" may be seen by relinking as before, but with a DRIVER.OBJ that has been assembled with the desired name:

```
ml /D _NAME_=PROTMAN$ driver.asm
```

Rebooting with the same CONFIG.SYS as before gives quite different results for a PROTMAN version of the test program:

```
Ceiling segment for conventional memory is 9000h

Ceiling segment for conventional memory is 9000h
```

It seems fair to say that the published literature on DOS programming is conspicuously void of thoughtful analyses of how to write device drivers, even ones like HIMEM.SYS that do not actually use the device driver interface for anything other than initialization. This author does not know why or how whole books have been written about device drivers without producing a satisfactory method for writing them in a high-level language, but a key liberation is to ignore Microsoft's frequently-expressed injunction that only a handful of DOS functions may be used during device driver initialization.

Sample Two—Loading an Arbitrary Program as a Device Driver

In case it is not sufficiently clear, the analysis presented in Chapter 7 shows that almost all properly functional DOS programs (including TSRs) available in object module format may be relinked so that they will also load as device drivers. Readers who have ever needed to write a serious device driver (or just to get code into memory before DOS is ready to execute programs) may appreciate that Chapter 7 is the best possible application of the knowledge gained about system startup and configuration in Part 1 of this book.

While on the subject of converting programs to load as device drivers, it is hard to resist pointing out a reversal of the common "DEVLOAD" exercise in which one tries to load a device driver after the system has already been configured. With the tools presented here, an arbitrary program may be executed before the system is ordinarily thought to be ready.

Again, it is easy to write one program that loads another. The SPOTLOAD program of Chapter 5, though suitable, is rather more complicated than required here.

```
/* SPAWN.C */
#include    <errno.h>
#include    <stdio.h>
#include    <process.h>

int main (int argc, register char **argv)
```

```
{
    register int exitcode;

    argv ++;
    argc --;
    if (argc == 0) {
          printf ("\nSPAWN: requires name of program to load\n");
          exitcode = -1;
    }
    else {
          exitcode = spawnv (P_WAIT, *argv, argv);
          if (exitcode == -1) perror ("\nSPAWN");
    }
    return (exitcode);
}
```

The first command line argument is interpreted as the name of a file to execute and the remaining arguments become those of the child. Compiling the source—call it SPAWN.C—and linking with the usual C startup code and run-time library produces an executable that can be tested or debugged like any other. The resulting program is practically useless as it stands, but becomes more useful when it is relinked with DRIVER.ASM and CRTDRVR.LIB:

```
link driver+crtdrvr.lib+spawn,spawn;
```

With this, SPAWN.EXE turns into a device driver capable of executing an arbitrary program somewhat earlier than is sometimes declared possible by writers who have presumably not considered the matter for themselves. The obvious candidate for premature execution is a debugger (for example, device=spawn.exe symdeb.exe). With this, readers may value the opportunity to wander through memory during system initialization, picking out features described in Chapter 3.

One slight "catch," but by no means an insuperable limitation, is that premature execution of a debugger requires the otherwise-unnecessary (at least for the purposes of this book) CRTDRVR library support for far memory allocation. As will be explained later in the chapter, CRTDRVR supplies device driver initialization code that simulates the regalia of process management—a PSP, environment, memory arena headers, and so forth.—before passing control to the startup code for the C run time. The usual steps taken by the C startup code to shrink the program's initial memory block affect only an artifact of the CRTDRVR interface and do not actually release memory in which to run a child process. Far memory allocation is supported only if other free memory blocks already exist when the device driver is loaded. In practice this is easy enough to arrange by giving DOS some upper memory to manage beforehand. A suitable CONFIG.SYS file for the exercise of running a debugger during system startup contains such lines as

```
device c:\dos\himem.sys
device c:\windows\emm386.exe noems ram=c800-efff
device a:\spawn.exe e:\tools\symdeb.exe
devicehigh c:\dos\smartdrv.sys 2048 1024
dos high umb
```

where SPAWN will be loaded into the low memory block after HIMEM and EMM386 but will in turn load SYMDEB into the newly-created block of upper memory. At this stage conventional and upper memory are not yet treated separately, so no fuss need be made about enabling the upper-memory link or setting a **loadhigh** allocation strategy. The exercise succeeds as long as SPAWN and its child program are loaded before the single UMB is allocated by SYSINIT to deal with a **devicehigh** directive.

The main point to running a debugger this early is to witness the system's construction directly. One object of specific interest is the temporary PSP used by SYSINIT:

```
-db 9077:0000
9077:0000  CD 20 01 90 00 9A F0 FE-1D F0 00 01 64 02 C5 40   M ....p~.p..d.E@
9077:0010  7C 90 C5 40 7C 90 00 00-00 00 00 FF FF FF FF FF   |.E@|...........
9077:0020  FF FF FF FF FF FF FF FF-FF FF FF FF 00 00 52 05   ..............R.
9077:0030  87 90 14 00 18 00 77 90-FF FF FF FF 00 00 00 00   ......w.........
9077:0040  05 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
9077:0050  CD 21 CB 00 00 00 00 00-00 00 00 00 00 00 00 00   M!K.............
9077:0060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
9077:0070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
```

Note in particular that this PSP has neither a parent (offset 16h) nor an environment (offset 2Ch); avoiding the consequences of the former is one reason why the CRTDRVR library constructs a separate PSP. Another object of interest is the buffer in which SYSINIT tokenizes the CONFIG.SYS file. The following extract offers several opportunities to confirm the parsing behavior described in Chapter 3:

```
-db 901d:0000 10109
901D:0000  0A 42 31 30 2C 30 0D 0A-51 34 34 2C 34 33 37 2C   .B10,0..Q44,437,
901D:0010  43 3A 5C 44 4F 53 5C 43-4F 55 4E 54 52 59 2E 53   C:\DOS\COUNTRY.S
901D:0020  59 53 0D 0A 44 43 3A 5C-44 4F 53 5C 48 49 4D 45   YS..DC:\DOS\HIME
901D:0030  4D 2E 53 59 53 00 0A 44-43 3A 5C 57 49 4E 44 4F   M.SYS..DC:\WINDO
901D:0040  57 53 5C 45 4D 4D 33 38-36 2E 45 58 45 00 4E 4F   WS\EMM386.EXE.NO
901D:0050  45 4D 53 20 52 41 4D 3D-43 38 30 30 2D 45 46 46   EMS RAM=C800-EFF
901D:0060  46 0D 0A 30 44 45 56 49-43 45 20 45 3A 5C 54 4F   F..0DEVICE E:\TO
901D:0070  4F 4C 53 5C 51 45 4D 4D-5C 51 45 4D 4D 33 38 36   OLS\QEMM\QEMM386
901D:0080  2E 53 59 53 20 4E 4F 45-4D 53 20 58 3D 42 30 30   .SYS NOEMS X=B00
```

```
901D:0090  30 2D 42 37 46 46 20 52-41 4D 3D 43 38 30 30 2D  0-B7FF RAM=C800-
901D:00A0  45 46 46 46 0D 0A 44 41-3A 5C 53 50 41 57 4E 2E  EFFF..DA:\SPAWN.
901D:00B0  45 58 45 20 45 3A 5C 54-4F 4F 4C 53 5C 53 59 4D  EXE E:\TOOLS\SYM
901D:00C0  44 45 42 2E 45 58 45 0D-0A 55 43 3A 5C 44 4F 53  DEB.EXE..UC:\DOS
901D:00D0  5C 52 41 4D 44 52 49 56-45 2E 53 59 53 20 31 30  \RAMDRIVE.SYS 10
901D:00E0  32 34 20 2F 45 20 0D 0A-55 43 3A 5C 44 4F 53 5C  24 /E ..UC:\DOS\
901D:00F0  53 4D 41 52 54 44 52 56-2E 53 59 53 20 32 30 34  SMARTDRV.SYS 204
901D:0100  38 20 31 30 32 34 20 0D-0A                       8 1024 ..
```

Each CONFIG.SYS command is reduced to a single letter: 'B' for **buffers**, 'Q' for **country**, and so forth. Each line is still terminated by the line-feed character taken directly from the text file. Look closely to see that SYSINIT imposes a null byte to separate a device driver's name from its command-line arguments in a **device** statement (introduced by the token 'D', as in the line for EMM386 at offset 0037h), but not for the new **devicehigh** command (beginning with the 'U' token, as in the line for RAMDRIVE at offset 00C9h). An exception is shown by SPAWN's own entry at offset 00A6h, in which the null separator has been converted temporarily to a space, this being the form expected when device drivers are passed their pathname and command-line arguments at initialization. Note finally that lines beginning with **rem** in CONFIG.SYS start with the '0' token in the configuration buffer, but are otherwise copied intact (as illustrated by the unwanted **device** statement for QEMM386 at offset 0063h).

Resident Programs

Real investigative power comes not with toys that take snapshots of the system in its half-formed state, but rather with programs that stay in memory, either to exert continuing control over the system's development or to start monitoring the system's configuration from as early a stage as possible.

The device driver techniques of Chapter 7 make it easy to arrange for a resident program to be installed sufficiently early. Development may take place in a more substantial environment than is available during system startup. In many cases, for instance, they may be tested inside a Windows virtual machine to give a greater chance of recovery (the DOS box with a misbehaving TSR can be terminated, and a fresh one started). Once the program is known to work, it may be relinked to load as a device driver. There remains the question of how to write a resident program efficiently in the first place.

The two strands (resident programming and device driver writing) come together in the construction of a real device driver, such as one that uses the device-driver interface to communicate with the DOS kernel after initialization. For this category it is especially clear that when code left in memory is reactivated by an interrupt of some kind, or by a device driver call from the DOS kernel, it will not expect to have access to the standard library functions. Many of the most powerful resident utilities perform their work in relative silence, having hooked one or more interrupts in order to enhance the services available to other applications. Not all TSRs have a pop-up stage in which they want to behave like a foreground process. There is a good case for implementing these as specialized task switchers which treat the pop-up code separately, allowing it to be swapped out to disk and reloaded when needed.

The emphasis must lie on finding a way to write resident programs that have all the machinery of the standard C library available during their installation but leave only the smallest possible footprint when they "go TSR." Again, this is a natural application of the knowledge gained about DOS program management in Part 1. A practical benefit is the development throughout Part 3 of a fairly substantial utility that maps extended memory usage but consumes just 2.5K of conventional memory—a small enough footprint to warrant its permanent installation as a diagnostic aid.

As elsewhere in this book, the tools developed here use Microsoft C, with very little thought given to other compilers such as Borland C++. For other compilers, there are different considerations for segmentation, linking order, and so on. The sort of things done in this section are necessarily compiler-dependent. In any case, what we are really doing here is using our knowledge (from Part 1) of DOS's configurability and startup machinery, to overcome some development problems. I am not spoon-feeding the reader with universal solutions, but trying to guide you through a suggested process of analysis.

CHAPTER 6

Resident Programming in C

One of the main obstacles to writing resident programs in a high-level language rather than assembler is that less control, if any, is offered over the large-scale structure of the program. For a normal application in a typical single-user environment it doesn't matter much that even a carefully optimized C program requires more memory than an assembly language version written with a fraction of the care. But a program that stays in memory uses a valuable system resource and must offer a good return on the investment.

The convenience of writing at a high level may lead to faster development, and the resulting source code will be accessible to more programmers. Size added for these reasons may be acceptable, but retaining several kilobytes of material needed only during initialization is wholly wasteful and can only detract from professional software. Not a few companies have traded very successfully on products that save less memory than it takes a C program to write "Hello, World" with the `printf` function.

Through a combination of command-line switches, pragmas, and the **_based** keyword, Microsoft C (from version 6.0 onward) allows some control over the segments into which it compiles functions and variables. This at least allows a distinction within the C program of the code and data that must be retained from the remainder that may be discarded. With a little juggling, it is actually possible to see this through to its logical conclusion and produce C programs with acceptably small memory footprints.

Ideally, all the resident material would be gathered at the base of the program. A typical structure would be

resident code
resident data
other code
other data

with data at the end of the program so that uninitialized variables need not add to the program's image on disk. This arrangement is generally not possible in high-level languages because some scheme is used to ensure that code appears before data. In Microsoft C, for instance, a **DOSSEG** comment in the run-time module CRT0.OBJ directs the linker to place all segments with class names ending in 'CODE' before any others, regardless of the order in which the linker has encountered segments in other files.

Thus, even if all the user's object files are assembly language modules defining segments in the preceding order, linking with the C run time will produce the unhelpful arrangement

resident code
other code
resident data
other data

The resident data and non-resident code have to be swapped somehow—if not in the executable on disk, then certainly in memory whenever the program is run. This may seem intractable, but in fact very few problems are insurmountable in the real-mode environment of DOS programs, so let's press ahead with the design of a segmentation model, keeping in mind the need for clear delineation of these four regions, two of which must be repositioned.

Segmentation

A scheme in which segments are marked as resident by assigning them to classes set aside for the purpose would be very convenient, because the linker would be able to gather segments into the four regions irrespective of object file order. That scheme cannot be implemented here because segment names and attributes must be consistent with the Microsoft C compiler's conventions—which allow no freedom over class names.

Given that all code segments generated by the compiler have the class name 'CODE', the best that can be arranged is that the very first object module should define segments in class 'CODE' that will be considered resident and then follow these with some form of label marking the start of the next region. All code segments generated in subsequent modules will automatically be regarded as non-resident (and liable to be exchanged at run time with the resident data).

Applying similar considerations to the program's data and noting that in the situations of interest the default code segment _TEXT and the default data segments in DGROUP may be assumed to be non-resident leads to the following include file:

```
; KEEPSEGS.INC

;   Segmentation model for a C program (TSR or device driver) that discards
;   code and data needed only for initialization.

;   Paragraph-aligned fence marking the start of the resident code region:

KEEP_BTEXT              SEGMENT PARA COMMON 'CODE'
KEEP_BTEXT              ENDS

;   Resident code segment:
```

```
;       The command line switch /NT KEEP_TEXT directs the Microsoft C
;       compiler to place all of a module's code in a segment KEEP_TEXT
;       with class 'CODE'. The present implementation of the Keep C scheme
;       provides no other segment for resident code.

KEEP_TEXT          SEGMENT WORD PUBLIC 'CODE'
KEEP_TEXT          ENDS

;   Paragraph-aligned fence for non-resident code segments:

NOKEEP_BTEXT                 SEGMENT PARA COMMON 'CODE'
NOKEEP_BTEXT                 ENDS

;   All other segments belonging to class 'CODE', such as the default code
;   segment _TEXT, will be placed here. Given DOSSEG ordering, this area
;   will also include those segments whose classnames end in 'CODE'.

;   Paragraph-aligned fence marking the start of the resident data region:

KEEP_BDATA                   SEGMENT PARA COMMON 'FAR_DATA'
KEEP_BDATA                   ENDS

;   Resident data segments:

;       Supplying the command-line switch /ND KEEP to the C compiler results
;       in the creation of the segments KEEP, KEEP_CONST, and KEEP_BSS with
;       class 'FAR_DATA', all addressed with respect to KEEP_GROUP. The
;       present implementation provides no other segments for resident data.

;       Note that the C compiler uses the COMM directive to declare
;       uninitialized data, leaving it to the linker to assign a position,
;       typically outside the executable's disk image and almost certainly
;       above the fence between resident and non-resident data. It seems
;       impossible to account in advance for the linker's assignment, and
;       therefore uninitialized data is not retained.

KEEP                         SEGMENT WORD PUBLIC 'FAR_DATA'
KEEP                         ENDS
KEEP_CONST                   SEGMENT WORD PUBLIC 'FAR_DATA'
KEEP_CONST                   ENDS
KEEP_BSS           SEGMENT WORD PUBLIC 'FAR_DATA'
KEEP_BSS           ENDS

KEEP_GROUP                   GROUP    KEEP, KEEP_CONST, KEEP_BSS
```

```
;   Paragraph-aligned fence for non-resident data:

NOKEEP_BDATA                SEGMENT PARA COMMON 'FAR_DATA'
NOKEEP_BDATA                ENDS

;   All other segments in the program, including those comprising the
;   default data group DGROUP, will be placed here.
```

KEEP_TEXT and KEEP_GROUP are essentially the default segments for resident material, equivalent to _TEXT and DGROUP, respectively. Indeed, the compiler will treat them as the defaults in any file compiled with the command line switches /ND KEEP /NT KEEP_TEXT, and it is recommended that source code be organized into resident and non-resident modules for compilation with or without these switches, respectively.

A single segment for resident code and a group for resident data gives a limit of 64K for each, which ought to suffice. Compilation of non-resident material in any memory model other than tiny is not hindered, as long as conflicts over segment names are avoided. In medium and large models, compiling a source file named KEEP.C generates the KEEP_TEXT segment as default code, which might not fit the programmer's intentions. For the purposes of this book, however, some further restriction is introduced to simplify the algorithm for exchanging the non-resident code and resident data regions. Although the non-resident code may be compiled in several segments, it will be assumed that the total does not exceed 64K.

Divisions within the program file are marked by paragraph-aligned **COMMON** segments. A label is identified as resident data not because its segment address matches KEEP_GROUP but because it lies at or above the KEEP_BDATA fence and below the next one. This is an important provision for the extensibility of the scheme. As will be demonstrated later, a programmer wanting to incorporate library material (for instance, for floating point emulation) in the resident part of a program will be able to do so if the library provides segments of its own. Definitions for the extra resident segments are simply placed between the relevant fences in KEEPSEGS.INC, safe in the knowledge that the routines provided for rearranging the segments will run correctly without alteration.

Rearrangement

Given that **DOSSEG** ordering prevents the linker from writing the executable with all resident material conveniently at the base of the program, a compromise has been struck in which it remains to exchange the resident data with the non-resident code. It might be thought possible to call some function to switch the two regions and consult the .EXE header to find and modify all the relocatable segment references which would have been resolved by the DOS program loader. The problem with this idea is that although all explicit references to segment addresses can be found and updated to account for the switch, the function must be called before any opportunity arises to use segment values without explicit reference. It would be unsafe, for instance, to allow a segment address to be stored in a variable for future use. There exists no reliable method of tracking down all variables used for this purpose and the stored value would be invalid after the swap. This extends, with slight subtlety, to segment addresses pushed onto the stack, say as part of a 32-bit return address from a far call, and to segment values passed to unknown procedures.

A little reflection along these lines and a quick scan of the startup source file CRT0DAT.ASM (supplied with Microsoft C) shows this first option to be unfeasible. The earliest opportunity for a user-supplied function to be called is as a C initializer; this is how the graphics library functions are initialized without a deliberate request from the programmer. The function for exchanging segments would have to be called as the first of these initializers, a situation which could indeed be verified by the function—and without relying on compiler-specific features whose future support seems unlikely. So far, so good. Before calling the user-supplied initializers, the C run-time installs a divide-by-zero trap and in small memory model fixes up some far pointers for the floating point emulator. These implicit references to the _TEXT segment, one of those to be moved, can of course be adjusted after the rearrangement. The scheme is clearly getting out of hand, however, requiring finely-detailed knowledge of the run-time's preparations, which may possibly change as the product evolves.

Exchanging the resident data and non-resident code segments is therefore not possible within the context of the C program. It must be done before the C run-time executes. The solution, possible only because the entry point to a Microsoft C program is accessible as a public label and because the Microsoft linker ignores redundant entry points, is to declare an alternative start address for the executable so that segments can be rearranged before passing control to the usual entry point.

The initialization code is spread over several modules and gathered into a library. Each module contains the segmentation model in its default form. Placing the library first in the linkage will therefore ensure both recognition of the new entry point and adoption of the segmentation model for the "Keep C" scheme. If the library is not linked first, then the first file to be linked must be an assembled version of the KEEPSEGS.INC file, presumably with extra segments introduced by the programmer. Details will be returned to, with examples, once the scheme has been developed fully.

```
; KEEPINIT.INC

;   This include file is common to the source files which initialize the
;   Keep C scheme for resident programming in C.

;   The segmentation model for the Keep C scheme is defined in the include
;   file KEEPSEGS.INC and must appear first to ensure correct ordering of
;   segments.
```

```
INCLUDE             keepsegs.inc

;   Initialization consists of rearranging segments in memory before passing
;   control to the program's usual entry point. The areas of memory to be
;   swapped are delineated by paragraph-aligned fences:

;        Before                                  After

;        KEEP_BTEXT                              KEEP_BTEXT
;            resident code                           resident code
;        NOKEEP_BTEXT                            KEEP_BDATA
;            other code                              resident data
;        KEEP_BDATA                              NOKEEP_BTEXT
;            resident data                           other code
;        NOKEEP_BDATA                            NOKEEP_BDATA
;            other data                              other data

;   The resident data will be swapped with the non-resident code. The
;   procedures responsible for this must be safely outside these regions,
;   which is why the Keep C initialization code is declared in a special
;   segment in the non-resident data area.

KEEPCINIT           SEGMENT WORD PUBLIC 'KEEPCINIT'
KEEPCINIT           ENDS

;   The default segments:

.MODEL                        SMALL, FARSTACK

;   Optional material offering general support for resident programming is
;   supplied in the KEEPCLIB library. An INCLUDELIB directive here saves
;   the user from specifying that library explicitly when linking the
;   program.

INCLUDELIB                    keepclib.lib

;   ==========================================================================

;   Structure for passing information about the program's segmentation:

KEEPINITINFO                  STRUCT

spOtherCode                   dw        ?
OtherCodeParas      dw        ?
```

```
OtherCodeSize          dw    ?

spKeepData                   dw        ?
KeepDataParas                dw        ?
KeepDataSize           dw    ?

spOtherData                  dw        ?

KEEPINITINFO                 ENDS
```

The parent file KEEPINIT.ASM contains the entry point and the transfer to the C run time. It calls procedures in other modules to swap the segments and to update the relocatable references from the .EXE header, allowing algorithm details to be hidden. The parent module must also be able to abort the program if the segments cannot be swapped safely.

It is important to realize that certain link options designed for optimizing large programs interfere fatally with the address calculations and segment repositioning. The /PACKCODE option, for instance, creates groups of code segments automatically in an attempt to increase the number of far calls that may be optimized away when the linker is invoked with the /FARCALLTRANSLATION switch. The offset component of a label's address will then operate with respect to the segment at the base of the group. If the label and the base are not both resident or both non-resident, then one will be moved but not the other. The label's address will be invalid after the segment rearrangement and the program must not continue.

Abortion will also be necessary if a valid .EXE header for the program cannot be loaded from disk, whether because of a file I/O error or an inability to obtain a pathname for the program. In this simplified version all such errors are reported identically.

```
; KEEPINIT.ASM

.NOLIST
  INCLUDE          ascii.inc
  INCLUDE          keepinit.inc
.LIST

;   ======================================================================

.CODE                        KEEPCINIT

InitKeepC        PROC        FAR C PUBLIC

                             EXTERN    SwapSegments:NEAR
```

```
                EXTERN  SwapReferences:NEAR
                EXTERN  C _astart:FAR

;   This is the entry point for an executable using the Keep C scheme for
;   resident programming. Segments are rearranged in memory before passing
;   control to the usual C run-time entry point defined in the CRT0 module.

;   Preserve those registers that are given some significance by the DOS
;   program loader—most important are the two segment registers ds and es,
;   which address the PSP, but note that the two bytes of register ax
;   describe the validity of the FCB fields in the PSP and are actually
;   consulted by some programs.

                push    es
                push    ds
                push    ax

;   Guard against linker options like /PACKCODE by checking that all fences
;   are in fact addressed in the form seg:0000.

                mov     ax,OFFSET NOKEEP_BTEXT
                or      ax,OFFSET KEEP_BDATA
                or      ax,OFFSET NOKEEP_BDATA
                jnz     abort

;   The other procedures involved in the segment swap need access to the
;   original segment addresses and to information about size. It is
;   convenient to pass both in the KEEPINITINFO structure. The present
;   implementation restricts the resident data and non-resident code to 64K
;   each (1000h paragraphs). Note also that were either area empty, the
;   swap would be superfluous.

                mov     ax,DGROUP
                mov     ds,ax
                mov     si,OFFSET KeepInitInfo
                ASSUME  si:PTR KEEPINITINFO

                mov     dx,NOKEEP_BTEXT
                mov     [si].spOtherCode,dx

                mov     bx,KEEP_BDATA
                mov     [si].spKeepData,bx

                mov     ax,NOKEEP_BDATA
                mov     [si].spOtherData,ax
```

```
                        sub     ax,bx
                        jz      done
                        cmp     ax,1000h
                        jnb     abort
                        mov     [si].KeepDataParas,ax
                        mov     cl,04h
                        shl     ax,cl
                        mov     [si].KeepDataSize,ax

                        sub     bx,dx
                        jz      done
                        cmp     bx,1000h
                        jnb     abort
                        mov     [si].OtherCodeParas,bx
                        shl     bx,cl
                        mov     [si].OtherCodeSize,bx

                        ASSUME  si:NOTHING

;   Swap the resident data with the non-resident code.

                        call    SwapSegments
                        jc      abort

;   Search the program's image in memory, adjusting the segment addresses
;   that were filled in from the .EXE header when the program was being
;   loaded.

                        call    SwapReferences
                        jc      abort
done:

;   Pass control to the C run-time as if nothing had ever happened!

                        pop     ax
                        pop     ds
                        pop     es
                        jmp     _TEXT:_astart
abort:

;   If for some reason, the segment rearrangement cannot be completed,
;   terminate the program.
```

```
                                mov     dx,OFFSET KeepInitError
                                mov     ah,09h
                                int     21h
                                mov     ax,4CFFh
                                int     21h

InitKeepC       ENDP

;   ======================================================================

.CONST

;   The single fatal error message:

KeepInitError                   db      CR, LF,
                                        "Keep C fatal error: ",
                                        "Unable to fixup segment references",
                                        CR, LF,
                                        "$"

;   ======================================================================

.DATA?

;   Structure for passing information about the program's segmentation:

                                PUBLIC  KeepInitInfo
KeepInitInfo                    KEEPINITINFO    <>

;   **********************************************************************

;   Direct the linker to make InitKeepC the entry point to the executable.

END                             InitKeepC
```

Swapping Segments

The easiest way to swap the non-resident code with the resident data is to obtain from DOS a block of memory into which the smaller region may be copied for temporary storage while the other region is moved into place. Looking to the future development of reasonably complex TSRs, this method is likely to seem extravagant in some situations and may even be unaffordable, if, for instance, the program has been loaded into a tight squeeze, perhaps at a time when DOS has no free memory blocks.

A better solution, requiring no external memory allocation, is to move material in pieces of some fixed size. If the regions involved in the move are both whole multiples of the transfer unit, then an efficient cyclical algorithm can be started by removing one unit temporarily. When the unit meant to occupy that space after the rearrangement is copied to its destination, a space will be left behind to be filled by a third unit, and so on. Readers with a mathematical background will know that if the numbers of units in the two regions are relatively prime, then the cycle will complete the swap. Otherwise, the algorithm requires some number of co-cycles. Either way the memory requirement is only a statically allocated buffer for temporary storage of the first unit in each cocycle; apart from that, each byte involved in the transfer is moved only once.

```
; SWAPSEGS.ASM

.NOLIST
  INCLUDE          keepinit.inc
.LIST

;   ========================================================================

.DATA?

;   A paragraph buffer used for the segment swap and a counter to determine
;   when the swap has been completed:

buffer                      db       10h DUP (?)
count                       dw       ?

;   Structure for obtaining information about the program's segmentation:

                            EXTERN   KeepInitInfo:KEEPINITINFO

;   ========================================================================

.CODE                       KEEPCINIT

SwapSegments                PROC     NEAR PUBLIC USES es

;   Resident data is swapped with non-resident code to gather resident
;   material at the base of the program. The algorithm employed in this
;   procedure avoids any possibly excessive memory allocation by moving data
;   a paragraph at a time, cycling through the transfer region (non-resident
;   code and resident data) until a complete swap has been effected.

;   Each paragraph is moved once, disregarding an occasional need to store a
;   paragraph temporarily. Counting the number of moves provides an easy
;   way to establish completion.

                                mov     dx,[KeepInitInfo.spOtherCode]
                                mov     ax,[KeepInitInfo.spOtherData]
                                sub     ax,dx
                                mov     [count],ax

;   Begin the cycle at offset zero in the non-resident code region. If
;   cocycles are necessary, each will be offset one paragraph from its
;   predecessor.
                                xor     ax,ax
next:

;   Save the starting address of the current cocycle.

                                mov     bx,ax

;   The first paragraph of code is copied to a temporary buffer, then
;   replaced by the first paragraph of data. A cycle is then established
;   in which paragraphs are moved into place as space becomes available—
;   the source of the previous transfer becomes the destination of the next.
;   The cycle closes when the address calculated as the next source is the
;   original location of the first paragraph of code.

                                push    ds
                                pop     es
                                mov     di,OFFSET buffer
copy:

;   At this stage, es:di will already have been pointed to the destination,
;   this being (in general) the source of the previous transfer. The new
;   source will have been calculated in dx:ax - load this into ds:si and
;   copy the data.

                                push    ds
                                mov     si,ax
                                mov     ds,dx
                                mov     cx,08h
                                rep     movsw
                                pop     ds
                                dec     [count]

;   Set the destination for the next transfer.
```

```
                              mov       di,ax
                              mov       es,dx

;   Compute the next source. This means advancing dx:ax by the size of the
;   non-resident code, but is complicated by segmentation and by the need to
;   check for completion of the cycle. Note that the present implementation
;   assumes that the non-resident code and resident data may each be
;   addressed with respect to a single segment, although together, they may
;   exceed 64K.

                              cmp       dx,[KeepInitInfo.spKeepData]
                              jnb       @f

;   If the previous source was in the code segment, the next source will
;   have the same offset but with respect to the data segment instead (since
;   the displacement = codesize). Wrap-around (back to the code segment
;   and requiring an adjustment of datasize) occurs if ax >= datasize.

                              cmp       ax,[KeepInitInfo.KeepDataSize]
                              jnb       wraparound
                              mov       dx,[KeepInitInfo.spKeepData]
                              jmp       copy
@@:

;   The source pointer (currently in the data region) is advanced by the
;   size of non-resident code, but note that wrap-around occurs if
;   ax + codesize >= datasize.

                              add       ax,[KeepInitInfo.OtherCodeSize]
                              jbe       @f
                              cmp       ax,[KeepInitInfo.KeepDataSize]
                              jb        copy
@@:
                              mov       dx,[KeepInitInfo.spOtherCode]
wraparound:
                              sub       ax,[KeepInitInfo.KeepDataSize]

;   Because the source for the next transfer lies in the code segment,
;   check for co-cycle completion before proceeding. If this is the last
;   paragraph in the co-cycle, fill the request from the temporary storage
;   buffer on the stack.

                              cmp       ax,bx
```

```
                                  jnz       copy
                                  mov       si,OFFSET buffer
                                  mov       cx,08h
                                  rep       movsw

;   If the number of paragraphs of non-resident code and resident data are
;   not relatively prime, then a complete swap requires more than one
;   co-cycle of paragraph transfers, each offset by one paragraph from the
;   one before.

                                  add       ax,10h
                                  cmp       [count],0000h
                                  jnz       next
                                  ret

SwapSegments                      ENDP

END
```

Swapping References

Far references to addresses in the two regions just swapped are now invalid and have to be updated before any attempted access. DOS itself is faced with a similar problem whenever it loads a .EXE file. To allow the program to be run at arbitrary locations in memory, the segment addresses in the program's image on disk are not absolute values—instead they are relative to the program's addressing base, awaiting resolution before the program receives control. As discussed in Chapter 5, DOS finds the segment references by inspecting a relocation table in the .EXE header. Now that the Keep C initialization has invalidated some of the segment addresses, it too must inspect the .EXE header in order to provide the correct references.

Reloading the .EXE header from disk requires first of all that the program can be located within the file system. The pathname used to load a program is entered automatically at the end of the program's environment in all versions of DOS since 3.0, so the technique is fairly well known.

```
; PROGNAME.ASM

.NOLIST
  INCLUDE         arena.inc
  INCLUDE         keepinit.inc
  INCLUDE         psp.inc
.LIST

;    ====================================================================
.CODE                   KEEPCINIT

GetProgramName          PROC    NEAR PUBLIC

;   This procedure expects AX to contain the segment address of a PSP. It
;   uses this to obtain the pathname with which that process was loaded from
;   disk, returning with a far pointer in dx:ax.

;   After the DOS "load and execute" function 4Bh has constructed an
;   environment for the new process, it appends the pathname whose address
;   was supplied as DS:DX. The word at offset 2Ch in the PSP is the segment
;   address of the process's environment, but note that it is possible to
;   load a program without an environment, in which case this word will be
;   zero.

                        mov     es,ax
                        xor     di,di

                        mov     dx,es:(PSP PTR [di]).spEnvironment
                        or      dx,dx
                        jz      badenv

;   The memory arena header immediately below the environment can be used
;   both to verify that the environment block belongs to the designated PSP
;   and to obtain the size of the environment.

                        dec     dx
                        mov     es,dx
                        cmp     es:(ARENA_HEADER PTR [di]).spOwner,ax
                        jnz     badenv

                        mov     cx,es:(ARENA_HEADER PTR [di]).wParagraphs
                        shl     cx,1
                        shl     cx,1
                        shl     cx,1
                        shl     cx,1

                        inc     dx
                        mov     es,dx

;   The environment consists of contiguous ASCIIZ strings, the last of which
;   is followed by another null byte.

                        xor     al,al
```

```
@@:
                        repnz   scasb
                        jnz     badenv

                        dec     cx
                        jle     badenv

                        scasb
                        jnz     @b

;   The pathname—another ASCIIZ string—should follow the word 0001h. If
;   this all fits within the environment block, return with dx:ax addressing
;   the pathname (and with a clear carry flag).

                        dec     cx
                        dec     cx
                        jle     badenv

                        mov     ax,0001h
                        scasw
                        jnz     badenv

                        mov     bx,di
                        xor     al,al
                        repnz   scasb
                        mov     ax,bx
                        jz      @f
badenv:

;   If for any reason, the pathname cannot be found, return a null pointer
;   in dx:ax and a set carry flag.

                        xor     ax,ax
                        mov     dx,ax
                        stc
@@:
                        ret

GetProgramName          ENDP

;   ************************************************************************

END
```

If the program's pathname can be found, then the .EXE header can be reloaded from disk, say a page at a time, to find those segment references that have to be updated to reflect the swap. One complication to all this juggling is the existence of schemes (such as linking with the /EXEPACK option) for reducing the size of executables, some of which move and compress the information contained in the relocation table, inserting code that will put everything right at run-time. The relocation table left behind is valid from DOS's point of view, but does not contain the information needed for the problem at hand.

Reliable detection is not easy, given the large number of schemes available. It may be felt sufficient to warn users of the incompatibility with the /EXEPACK option and with file packing utilities in general. In this context, observe that the Windows 3.0 version of DOSX.EXE, which reloads its .EXE header in order to replace real-mode segments by protected-mode selectors, accepts whatever .EXE header it finds and will not work if the .EXE was packed. The scheme being developed here for resident programming is intended to have more generality, however, so some checking ought to be implemented.

A second complication centers on symbolic debugging. Many programmers feel that debugging is difficult enough, without having a supposedly helpful segment rearrangement invalidate the debugger's symbol tables. Fortunately, Windows also moves segments around in memory, so the problem of debugging in this type of situation has been attended to already. Details will be taken up shortly—the important point for the moment is that the procedure for updating the segment references should be written with the debugging problem in mind.

```
; SWAPREFS.ASM

.NOLIST
  INCLUDE          exehdr.inc
  INCLUDE          keepinit.inc
.LIST

;    ======================================================================

.DATA?

;    A 0200h byte buffer to hold a page from the program's EXE header—
;    preceded by a few bytes to anticipate relocation entries that are split
;    between two pages:

                                  db        04h DUP (?)
ExePageBuffer                     db        0200h DUP (?)

;    Structure for obtaining information about the program's segmentation:

                                  EXTERN    KeepInitInfo:KEEPINITINFO
```

```
;   ===========================================================================

.CODE                           KEEPCINIT

SwapReferences                  PROC      NEAR PUBLIC USES es

                                EXTERN    GetProgramName:NEAR

;   Check for the presence of a debugger supporting the SEGDEBUG interface—
;   if present, it may be alerted to the segment rearrangements.

IFDEF                           _SEGDEBUG_
                                EXTERN    FindSegdebug:NEAR
                                call      FindSegdebug
ENDIF

;   Obtain a pathname for the program file. The procedure which does this
;   expects the PSP segment address in register AX, returning with dx:ax
;   pointing to the pathname.

                                mov       ax,es
                                call      GetProgramName
                                jc        fail

;   Open the file. Keep the file handle in register bx for the remainder
;   of the procedure.

                                push      ds
                                xchg      ax,dx
                                mov       ds,ax
                                mov       ax,3D00h
                                int       21h
                                pop       ds
                                jc        fail
                                mov       bx,ax

;   Attempt to read the first 512 bytes of the file. Establish that the file
;   is large enough to contain an EXE header and that it does in fact begin
;   with an EXE signature.

                                mov       ah,3Fh
                                mov       cx,0200h
                                mov       dx,OFFSET ExePageBuffer
                                int       21h
```

```
                             jc      close
                             cmp     ax,TYPE EXEHDR
                             jb      close
                             mov     si,dx
                             mov     ax,(EXEHDR PTR [si]).wSignature
                             cmp     ax,5A4Dh   ; 'MZ'
                             jz      @f
                             cmp     ax,4D5Ah   ; 'ZM' too!
                             jnz     close
@@:

;   Prepare to examine the EXE header's relocation table. There must be at
;   least some entries in the relocation table (this source file generates
;   one), which provide at least a cursory check that the program has not
;   been linked with the /EXEPACK option.

                             mov     cx,(EXEHDR PTR [si]).wRelocations
                             jcxz    close
                             add     si,(EXEHDR PTR [si]).pRelocationTable
next:

;   Read the relocation entries one by one until there are none left. Each
;   entry is a far pointer offset from the program's address frame. The
;   details of retrieving an entry are handled in a separate procedure, which
;   decrements cx and advances si, reading a new page from the EXE header
;   if necessary. The entry itself is returned in ax:di.

                             jcxz    done
                             call    GetEntry
                             jc      close

;   Convert the entry to a far pointer and correct the address if it refers
;   to non-resident code or resident data, since these will have been moved.

                             add     ax,KEEP_BTEXT
                             call    AdjustSegment

;   If the target of the segment reference is one of those which has been
;   moved, then update it.

                             mov     es,ax
                             mov     ax,es:[di]
                             call    AdjustSegment
                             jnc     next
                             stosw
```

```
                                jmp     next
close:
                                mov     ah,3Eh
                                int     21h
fail:
                                stc
                                jmp     @f
done:
                                mov     ah,3Eh
                                int     21h
@@:
                                ret

SwapReferences                  ENDP

;   ----------------------------------------------------------------

GetEntry            PROC        NEAR PRIVATE

                                push    cx
                                mov     dx,OFFSET ExePageBuffer

;   In most cases, SI will address the relocation entry which this procedure
;   is intended to return. The entry will be taken from a buffer containing
;   a page (0200h bytes) from the EXE header. If there are at least four bytes
;   left in the buffer, then read the entry into ax:di and return, having
;   advanced si and decremented cx. If no space is left in the buffer, then
;   load a new page, reset si to the start of the buffer, and continue as
;   before.

                                mov     cx,OFFSET ExePageBuffer + 0200h
                                sub     cx,si
                                cmp     cx,04h
                                jnb     read_entry
                                or      cx,cx
                                jnz     split_entry
                                mov     si,dx
                                jmp     load_page
split_entry:

;   If the relocation entry is split between two pages, copy the remaining
;   bytes to the area left immediately in front of the buffer (for precisely
;   this purpose), before going on to load a new page.

                                push    ds
```

```
                        pop     es
                        mov     di,dx
                        sub     di,cx
                        mov     ax,di
                        rep     movsb
                        mov     si,ax

load_page:
                        mov     ah,3Fh
                        mov     cx,0200h
                        int     21h
                        jc      @f

read_entry:
                        lodsw
                        mov     di,ax
                        lodsw
                        clc
@@:
                        pop     cx
                        dec     cx
                        ret

GetEntry        ENDP

;   ------------------------------------------------------------

AdjustSegment           PROC    NEAR PRIVATE

;   Return a clear carry flag unless AX is in the region containing segments
;   that have been swapped, i.e., unless

;       original NOKEEP_BTEXT <= AX < original NOKEEP_BDATA

                        cmp     [KeepInitInfo.spOtherCode],ax
                        ja      done
                        cmp     ax,[KeepInitInfo.spOtherData]
                        jnb     done

;   Before correcting the segment value, save the original in dx for the
;   debugger.

IFDEF                   _SEGDEBUG_
                        mov     dx,ax
ENDIF
```

```
;   Non-resident code has been moved upward to make way for the resident
;   data.
                              cmp     ax,[KeepInitInfo.spKeepData]
                              jnb     @f
                              add     ax,[KeepInitInfo.KeepDataParas]
                              jmp     set
@@:

;   Resident data has been moved downward.

                              sub     ax,[KeepInitInfo.OtherCodeParas]
set:

;   Tell the debugger about the change in segment value so that it may
;   update its symbol tables (pass ax = new value, dx = old value).

IFDEF                         _SEGDEBUG_
                              EXTERN  AlertSegdebug:NEAR
                              call    AlertSegdebug
ENDIF
                              stc
done:
                              ret

AdjustSegment                 ENDP

END
```

Debugging

The author's favorite real-mode debugger—the long discontinued SYMDEB—provides an interface through which its symbol tables may be updated, usually by Windows whenever it loads, moves, or discards a program segment. Whether or not the service was implemented specifically to help debug Windows applications, it has been designed as a general interface. As an aside, observe that debuggers are not the only programs with an interest in segment relocations. The profiler PROF.COM distributed with the Windows 3.0 SDK also understands this interface.

No doubt, other debuggers such as CodeView provide similar interfaces to cope with segment movements, but documentation is scarce, so the author has chosen to support only the debugger he knows and happens to prefer. Of course, the support has been framed in a way that enables easy replacement should the reader feel the need. The module is not needed at all unless the _SEGDEBUG_ symbol is defined while assembling SWAPREFS.ASM.

```
; SEGDEBUG.ASM

;   This file is required for the debug version of the CRTKEEPC library. It
;   checks for the presence of a SEGDEBUG interface and uses this to inform
;   the debugger about the segment rearrangements.

.NOLIST
 INCLUDE        keepinit.inc
.LIST

;   ==========================================================================

.CONST

;   Identification string:

Segdebug        db        "SEGDEBUG", 00h

;   ==========================================================================

.DATA?

;   The address at which to call the debugger:

SegdebugEntry             dd        ?

;   A list of segments whose repositioning has already been conveyed and a
;   pointer into the list:

SegmentList               dw        40h DUP (?)        ; an arbitrary number
NextEntry       dw        ?

;   ==========================================================================

.CODE                     KEEPCINIT

FindSegdebug              PROC      NEAR PUBLIC USES es ds di si

;   If the int 03h trap points to a segment containing the SEGDEBUG
;   identification string at offset 0100h, then the debugger's entry point
;   is at offset 00FBh in that same segment. Codeview operates similarly,
;   using the identification string CVDEBUG.

                        xor     ax,ax
                        mov     dx,ax
                        mov     es,ax
                        mov     es,es:[03h * 4 + 02h]
                        mov     di,0100h
                        mov     si,OFFSET Segdebug
                        mov     cx,SIZEOF Segdebug
                        repz    cmpsb
                        jnz     @f

                        mov     ax,OFFSET SegmentList
                        mov     [NextEntry],ax

                        mov     ax,00FBh
                        mov     dx,es
@@:
                        mov     word ptr [SegdebugEntry],ax
                        mov     word ptr [SegdebugEntry + 02h],dx
                        ret

FindSegdebug            ENDP

;   -----------------------------------------------------------------------

AlertSegdebug           PROC    NEAR PUBLIC USES es di dx cx bx ax

;   If the SEGDEBUG interface is present, then notify it that the segment DX
;   will now be addressed by AX, unless it has been informed already.

;   All registers are preserved. Note that the SEGDEBUG call preserves
;   registers si, di, bp, ds, and es.

                        cmp     word ptr [SegdebugEntry],0000h
                        jz      done

                        mov     cx,[NextEntry]
                        cmp     cx,OFFSET SegmentList + SIZEOF SegmentList
                        jnb     done

                        xchg    ax,dx

                        push    ds
                        pop     es
```

```
                             mov       di,OFFSET SegmentList
                             sub       cx,di
                             jz        @f

                             shr       cx,1
                             repnz     scasw
                             jz        done
@@:
                             stosw
                             mov       [NextEntry],di

;   The SEGDEBUG call takes arguments on the stack. In the language of C
;   prototypes, this is Segdebug (1, oldseg, newseg).

                             push      dx
                             push      ax
                             mov       ax,0001h
                             push      ax
                             call      [SegdebugEntry]
                             add       sp,06h
done:
                             ret

AlertSegdebug                ENDP

END
```

Support

The final steps before terminating a program that will stay in memory usually center on a few small clean-up tasks, such as closing file handles, that DOS performs automatically for programs terminating by the usual means. When a program exits with DOS function 31h (Terminate and Stay Resident) or the obsolete int 27h, DOS does not reclaim system resources. It resizes the program's main memory block to avoid the prospect of executing code in unallocated memory, but with only that exception DOS assumes that the TSR will have surrendered any resources it does not expect to need again.

Releasing System Resources

Once initialized, many TSRs have no need for their file handles or environment block and ought therefore to close the handles and release the environment. Functions for performing these operations and others will have a general usefulness, suggesting that they be provided in a library offering general support for resident programming in C. This library KEEPCLIB.LIB is the one named earlier in an **INCLUDELIB** directive for the convenience of not having to link it explicitly. The price to pay for binding the libraries this way is that either KEEPCLIB should be model-independent or that both libraries be available in all memory models. For the instructional purposes of this book it will suffice to work with small model.

Functions to close file handles and release the environment are written easily enough in C.

```
/* CLOSEALL.C */

#include    <dos.h>
#include    <stdlib.h>

#include    <psp.h>
#include    <standard.h>

void CloseAllFiles (void)
{
    /*   This function closes all file handles.Note that there is no
         way to determine at DOS-function level the number of open
         handles or even a useful upper bound. Ordinarily, programs
         written in Microsoft C have a maximum of 20 handles, but the
         user is provided with instructions for increasing this by
         modifying the C run-time module CRT0DAT.

         Each process has its own handle table, the address and size of
         which may be found by inspecting the PSP. The table consists of
         single-byte indexes into the System File Tables. The entry FFh
         indicates that no open file is associated with the handle. The
         Microsoft C library function _dos_close is called to close each
         open handle. */

    register int handle;
    _segment tableseg;
    register BYTE _based (tableseg) *tableptr;

    #define pspaddr ((struct PSP _far *) MK_FP (_psp, NULL))

    handle = (pspaddr -> wNumberOfHandles);
    if (handle) {
         handle --;
         tableptr = (pspaddr -> lpHandleTable) + handle;
         tableseg = (_segment) (pspaddr -> lpHandleTable);
         do {
             if (*tableptr != 0xFF) _dos_close (handle);
```

```
            tableptr --;
            handle --;
        } while (handle >= 0);
    }
}


/* FREE_ENV.C */

#include    <dos.h>
#include    <stdlib.h>

#include    <psp.h>
#include    <standard.h>

unsigned FreeEnvironment (void)
{
    /*    This function calls DOS to release the block of memory beginning
          with the program's environment. It returns a DOS error code if
          the environment exists but cannot be released;  0, otherwise.

          The environment is located by consulting the program's PSP, the
          segment address of which is recorded by the C run-time and made
          available through the variable _psp. Memory is released by
          calling the Microsoft C library function _dos_freemem (). */

    _segment envseg;
    unsigned errorcode;

    #define pspaddr ((struct PSP _far *) MK_FP (_psp, NULL))

    envseg = pspaddr -> spEnvironment;
    if (envseg) {
          errorcode = _dos_freemem (envseg);
          if (errorcode == 0) pspaddr -> spEnvironment = 0;
          else return (errorcode);
    }
    return (0);
}
```

Terminating and Staying Resident

As already noted, the TSR must pass DOS the amount of memory it wants to retain in its main program block. This is the block that begins with the PSP, so effectively the amount is not an absolute value but is instead an amount above the PSP. The preferred method for calculating the number of paragraphs expected by both DOS function 31h and its high-level equivalent `_dos_keep` in the Microsoft C library is to first decide on a break address, align to a paragraph boundary, and subtract the segment address made available in the variable `_psp`. This observation will be used to great effect in Chapter 7 when it is shown that any TSR relying on this method may be converted so that it may also load as a device driver.

In the meantime it remains to provide programs with a method for determining a suitable break address. Ordinarily, a program written in C would have to retain its initialization code, data, stack, and perhaps even some of the near heap. If the goal is economic coding without resorting to in-line assembler, then making a dummy allocation from the near heap provides a neat method of obtaining a usable break address. With the Keep C segmentation model, however, programs have a default break address marked by the fence segment NOKEEP_BTEXT. Access to this, at least in a public variable, will be needed as a base for programs to use in calculating the amount of memory required for uninitialized buffers.

Many programs will need no more memory than provided by the resident Keep C segments and should be provided with the means to accept the default without further trouble. Perhaps a more compelling reason for providing a new function for TSR termination is to allow programs to build tables over their initialization code before going resident. This cannot be done safely if terminating with the `_dos_keep` function. The integrity of that function and the high-level exit procedures it calls cannot be guaranteed after the table construction.

The solution is to implement a system of resident pre-terminators with a registration scheme along the lines provided by the `atexit` and `onexit` functions in the standard C library. Each pre-terminator must be a **FAR** procedure in a resident code segment and will be called at a time when the non-resident segments are no longer needed (and may therefore be overwritten safely).

```
; KEEPEXIT.ASM

.NOLIST
  INCLUDE         keepsegs.inc
  INCLUDE         psp.inc
.LIST

.MODEL                        SMALL, FARSTACK

;    ==================================================================

.DATA?

;    An uninitialized array in which to store addresses of functions
;   registered by AtExitKeepC or OnExitKeepC:

                            PUBLIC    XKEEPC_start, XKEEPC_end

XKEEPC_start                dd        10h DUP (?)
XKEEPC_end                  LABEL     DWORD

;   ======================================================================

.DATA

;   A pointer into the array XKEEPC_start:

                            PUBLIC    XKEEPC_ptr
XKEEPC_ptr                  dw        OFFSET XKEEPC_start

;   Make the default break address available publicly.

                            PUBLIC    C lpBreakAddress
lpBreakAddress              dd        NOKEEP_BTEXT:0000h

;   The amount of memory for retention is calculated with respect to the
;   PSP's segment address—supplied by the C run-time:

                            EXTERN    C _psp:WORD

;   ======================================================================

.CODE

ExitKeepC         PROC      C PUBLIC retcode:WORD, breakaddr:DWORD

IF                          @CodeSize
                            EXTERN    C _cexit:FAR
ELSE
                            EXTERN    C _cexit:NEAR
ENDIF

;   int _cdecl ExitKeepC (unsigned, void _far *)

;   Before examining the caller's requested break address, check the amount
;   of memory available in the program's main block. This is not done just
;   for defensiveness—it is the programmer's responsibility to ensure that
;   the requested break address is sensible—but because resident
;   pre-terminators are called from code relocated to the top of memory
;   available to the program.

;   DOS function 4Ah, if passed an impossible value in register bx, will
;   return with bx corrected to the largest valid size in paragraphs. That
;   it also expands the block even though the function fails is the subject
;   of one of those interminable debates about the difference between bugs
;   and features.
                                mov     ah,4Ah
                                mov     bx,0FFFFh
                                mov     es,[_psp]
                                int     21h

;   Update the memory ceiling field in the PSP.

                                mov     ax,es
                                add     ax,bx
                                mov     es:[PSP.spCeiling],ax

;   Retention just of the Keep C code and data segments may be specified by
;   supplying a null pointer for the break address. If the break address is
;   not aligned to a paragraph boundary, advance it to the next one. In all
;   cases, arrange to have the break segment (the segment address of the
;   lowest unneeded paragraph) in register dx.

                                mov     ax,word ptr [breakaddr]
                                mov     dx,word ptr [breakaddr + 02h]
                                or      ax,ax
                                jnz     normalise
                                or      dx,dx
                                jnz     @f
                                mov     dx,NOKEEP_BTEXT
                                jmp     @f
normalize:
                                add     ax,0Fh
                                rcr     ax,1
                                shr     ax,1
                                shr     ax,1
                                shr     ax,1
                                add     dx,ax
@@:

;   Perform the normal (non-resident) cleanup. This includes any user
;   terminators registered with the usual atexit or onexit functions.
;   Preserve the break segment across the call.
```

```
                push    dx
                call    _cexit
                pop     dx

;   To support programs that need to overwrite their initialization code
;   with data before terminating (as opposed to programs which want merely
;   to reserve the memory), the AtExitKeepC and OnExitKeepC functions are
;   provided to register as many as 16 (an arbitrary number) other functions
;   in a resident code segment to be called from safe memory before the
;   program terminates. Matters are much easier if no such functions
;   have been registered.

                mov     di,[XKEEPC_ptr]
                cmp     di,OFFSET XKEEPC_start
                jnz     @f

;   Subtract the PSP's segment address and use DOS function 31h to
;   terminate.

                sub     dx,[_psp]
                mov     al,byte ptr [retcode]
                jmp     terminate
@@:

;   The safety of the preterminators is guaranteed (they lie in a resident
;   segment), but it is possible that the address table and even the stack
;   may be overwritten if, for example, the program wants to clear a very
;   large data array before terminating.

;   Whether the table and stack need to be relocated or not, the procedure
;   for calling the preterminators expects es:si to address the table's
;   base and es:di to provide an upper bound (which it does already!).

                push    ds
                pop     es
                mov     si,OFFSET XKEEPC_start

;   Since the stack may be reset, retrieve the program's exit code.

                mov     bx,[retcode]

;   Remember that the lowest unneeded paragraph is in dx. If this is below
;   or equal to the paragraph in which the address table begins, then the
;   table and stack will be safe.
```

```
                                mov     ax,OFFSET XKEEPC_start
                                mov     cl,04h
                                shr     ax,cl
                                mov     cx,ds
                                add     ax,cx
                                cmp     dx,ax
                                jbe     ok

;   If the caller wants to overwrite so much memory before going resident
;   that there isn't at least a little room (400h bytes) for the final
;   manipulations, something is clearly wrong, because at the very least there
;   would be no need to use the Keep C scheme.  In such a case, return under
;   the hopeful assumption that the caller will know what to do. This is
;   the only reason for ExitKeepC to return.

                                mov     es,[_psp]
                                mov     ax,es:[PSP.spCeiling]
                                sub     ax,dx
                                jb      noexit
                                cmp     ax,400h SHR 4
                                jnb     @f
noexit:
                                xor     ax,ax
                                ret
@@:

;   Set up a new stack before moving the address table.

                                cmp     ax,1000h
                                jb      @f
                                mov     ax,0FFFh
@@:
                                shl     ax,cl

                                mov     es,dx
                                cli
                                mov     ss,dx
                                mov     sp,ax
                                sti

;   Copy the address table. Note that it is being moved upward in memory
;   and, just conceivably, the source and destination may overlap, so copy
;   from top to bottom. The new table will begin at offset zero in the new
;   stack segment. As before, es:si must address the table's base and es:di
;   its upper bound.
```

```
                        mov     cx,di
                        sub     cx,si
                        mov     si,di
                        dec     si
                        dec     si
                        mov     di,cx
                        push    di
                        dec     di
                        dec     di
                        shr     cx,1
                        std
                        rep     movsw
                        cld
                        pop     di
                        xor     si,si
ok:

;   Push the two arguments required by DOS function 31h onto the stack—the
;   number of paragraphs first, then the program's exit code.

                        sub     dx,[_psp]
                        push    dx
                        push    bx

;   Append the termination code to the address table--wherever the table is,
;   the memory immediately after it should be safe.

                        push    di
                        push    si
                        push    cs
                        pop     ds
                        mov     si,OFFSET moveable_exit
                        mov     cx,OFFSET endexit - OFFSET moveable_exit
                        rep     movsb
                        pop     si
                        pop     di

;   Set ds to the default group for resident data and transfer control to
;   safe memory.

                        mov     ax,KEEP_GROUP
                        mov     ds,ax
                        ASSUME  ds:KEEP_GROUP
                        push     es
                        push     di
                        retf

moveable_exit:

;   Code from here onward is executed after relocation away from material
;   that may be overwritten by the preterminators.

;   Although it is normally the responsibility of the _cexit cleanup code
;   to ensure that a Ctrl-C or Ctrl-Break pending at the time of the final
;   DOS call (function 31h) will be returned harmlessly, there is a problem
;   here in that the exception handlers (in a non-resident code segment) may be
;   overwritten. Thus, install temporary int 23h and int 24h handlers in
;   safe memory. The int 24h handler would be required for a resident
;   preterminator that attempts file I/O.

                        mov      ax,3300h
                        int      21h
                        push     dx

                        xor      dx,dx
                        mov      ax,3301h
                        int      21h

                        push     ds
                        push     cs
                        pop      ds
                        mov dx,OFFSET int23h_atexit - OFFSET moveable_exit
                        add      dx,di
                        mov      ax,2523h
                        int      21h
                        pop      ds

                        pop      dx
                        mov      ax,3301h
                        int      21h

                        push     ds
                        mov dx,OFFSET int24h_atexit - OFFSET moveable_exit
                        add      dx,di
                        mov      ax,2524h
                        int      21h
                        pop      ds
```

```
;   Call the functions in the address table. As with the preterminators
;   registered through the usual atexit and onexit functions, these are
;   called in a last-in, first-out order. The preterminators are expected
;   to be far functions lying in a resident code segment.

@@:
                                  sub       di,04h

                                  push      es
                                  call      dword ptr es:[di]
                                  pop       es

                                  cmp       si,di
                                  jb        @b

;   Retrieve the paragraph and exitcode values from the stack, then call DOS
;   function 31h to terminate.

                                  pop       ax
                                  pop       dx
terminate:
                                  mov       ah,31h
                                  int       21h

;   The simplest possible int 23h and int 24h handlers—fail all critical
;   errors and respond to Ctrl-C by restarting the "broken" DOS call

int24h_atexit:
                                  mov       al,03h
int23h_atexit:
                                  iret
endexit:

ExitKeepC           ENDP

END
```

Registering the preterminators is easy compared to arranging for them to be called. The relevant code could be written in C, but there is a small advantage to limiting the scope of labels. Without the leading underscore attached to global labels in C programs, objects like the address table for resident preterminators lie outside the reach of modules written in C. On a larger scale readers may have noticed that procedures are always declared **PRIVATE** unless they *must* be visible outside their module and that data is only declared **PUBLIC** if it *needs* to be. In general it is easier to retain in human memory the details of a project that has been properly modularized, with as few global references as possible.

```
; ONEXIT.ASM

.MODEL                      SMALL, FARSTACK

;   ===========================================================================

.DATA?

;   The table in which function addresses will be recorded is defined in the
;   module containing ExitKeepC, together with a pointer to the location at
;   which to make the next entry.

                            EXTERN   XKEEPC_start:DWORD, XKEEPC_end:DWORD
.DATA
                            EXTERN   XKEEPC_ptr:WORD

;   ===========================================================================

.CODE

AtExitKeepC                 PROC     C PUBLIC USES di, func:DWORD

;   Two forms of the function are supported, corresponding to the atexit
;   and onexit functions provided in the standard C library. They are
;   distinguished by the value returned on failure; apart from this, they
;   may share the same code.

                            OPTION   NOSCOPED

                            mov      cx,0FFFFh
                            jmp      @f

AtExitKeepC                 ENDP

OnExitKeepC                 PROC     C PUBLIC USES di, func:DWORD

                            xor      cx,cx
@@:

;   Check that space in the table has not been exhausted.

                            push     ds
```

```
                        pop     es
                        mov     di,[XKEEPC_ptr]
                        cmp     di,OFFSET XKEEPC_end
                        jz      fail

;   Enter the function's address. The opportunity exists to check that the
;   address lies in a resident code segment, but is not taken in this simple
;   version.

                        mov     ax,word ptr [func]
                        mov     dx,word ptr [func + 02h]
                        stosw
                        xchg    ax,dx
                        stosw
                        xchg    ax,dx

                        mov     [XKEEPC_ptr],di

;   The onexit form (cx = 0000h) indicates success by returning the function
;   address. The atexit form returns zero.

                        jcxz    done
                        xor     ax,ax
                        jmp     done
fail:
                        mov     ax,cx
                        mov     dx,cx
done:
                        ret

OnExitKeepC             ENDP

END
```

Interrupt Control

Unless the TSR has been designed to be inert after installation, perhaps to consume memory for testing, it must install some method of regaining control. A variety of methods are available, with different degrees of exposure. Interception of a hardware interrupt, for instance, may have a very broad effect on the system, but passing the DOS task switcher a calling address is a rather more private arrangement.

In the case of interrupts, whether hardware or software, it is essential to be able to pass control to other interested parties, most particularly (but not always), the handler from which the TSR wrested control of the interrupt. The Microsoft C library supplies a function `_chain_intr` for this purpose, but it lies in the _TEXT segment and is not retained. Therefore the Keep C library provides a replacement in the resident code segment KEEP_TEXT.

```
; CHAININT.ASM

.NOLIST
  INCLUDE         cpu.inc
  INCLUDE         intpack.inc
  INCLUDE         keepsegs.inc
.LIST

.MODEL                    SMALL, FARSTACK

;    ==========================================================================

.CODE                     KEEP_TEXT

ChainIntr         PROC    FAR C PUBLIC

;    void _far _cdecl ChainIntr (void (_cdecl _interrupt _far *) ())

;    This function replaces the Microsoft C library function _chain_intr
;    with more or less equivalent code in the resident KEEP_TEXT segment.
;    Like the usual function, it assumes that SS:BP addresses the registers
;    stacked by the prologue code generated by the _interrupt keyword. This
;    means that it must be called from the interrupt handler itself, not a
;    subordinate function.

;    When this procedure receives control, the stack will be laid out as
;    follows (from top to bottom):

;           the interrupt caller's flags (word)
;           return address to interrupt caller (dword)
;           stacked registers ax, cx, dx, bx, ... , ds, es (addressed by BP)
;           local variables declared in the interrupt handler (unknown number)
;           the transfer address passed as argument to this function (dword)
;           return address from this function (dword)

;    The aim is to transfer control to the address given as the argument to
;    this function, having first restored the registers from the stack. The
;    handler to which control is passed should find both the interrupt and
;    trap flags clear, just as it would have done had it been first in the
;    chain.
;   The method adopted is to prepare the flags and transfer address on the
;   stack in the position occupied by ax, cx, and dx, then, after popping all
;   the other registers from the stack, an iret instruction will transfer
;   control correctly.

;   Note: The following depends on PUSHA order (used by Microsoft but not
;   by Borland).

                                ASSUME  bp:PTR INTPACK

                                pop     ax
                                pop     ax

                                mov     dx,[bp].stacked_dx
                                pop     [bp].stacked_dx

                                mov     cx,[bp].stacked_cx
                                pop     [bp].stacked_cx

                                mov     ax,[bp].stacked_flags
                                and     ax,NOT (MASK IFbit OR MASK TFbit)
                                xchg    ax,[bp].stacked_ax

                                mov     sp,bp

                                pop     es
                                pop     ds
                                pop     di
                                pop     si
                                pop     bp
                                pop     bx
                                pop     bx

                                iret

ChainIntr           ENDP

END
```

Avoiding the Compiler's Library Calls

Occasionally it may be necessary to adopt some unusual practice to prevent the compiler calling functions in a non-resident segment. This consideration arises particularly with stack-checking, for which the compiler generates a call to `__aNchkstk` in small memory model (for example) and with arithmetical operations on 32-bit objects, which are often passed to library functions in the _TEXT segment. The first of these is dealt with easily enough by compiling resident modules (if not the whole program) with the /Gs switch, but the second may require some thought, as illustrated by the following functions.

```
/* SHIFT.C */

/*  This file contains functions which perform 32-bit shifts by operating
    separately on 16-bit components to avoid internal calls to non-resident
    library functions. */

#include    <standard.h>

/*  Functions are provided for shifting left and right. Small shifts
    operate separately on the high and low words, but, in addition, a piece
    of one word moves to the other. Shifts of more than 16 bits necessarily
    obliterate one component. */

DWORD _cdecl shl (DWORD x, WORD cl)
{
    WORD high, low, middle;

    if (cl < 16) {
          high = HIGHWORD (x) << cl;
          low = LOWWORD (x) << cl;
          middle = LOWWORD (x) >> 16 - cl;
          return (MK_DWORD (high | middle, low));
    }
    else return (MK_DWORD (LOWWORD (x) << cl - 16, 0));
}

DWORD _cdecl shr (DWORD x, WORD cl)
{
    WORD high, low, middle;

    if (cl < 16) {
          low = LOWWORD (x) >> cl;
          high = HIGHWORD (x) >> cl;
          middle = HIGHWORD (x) << 16 - cl;
          return (MK_DWORD (high, middle | low));
    }
    else return (MK_DWORD (0, HIGHWORD (x) >> cl - 16));
}
```

Some standard library functions can be coded in-line, avoiding the usual function call to a non-resident segment. The functions concerned include some that are used commonly, such as `strlen` and `memcpy`, or ones whose nature makes them useful at low-level, such as `_enable` and `inp`. The switch /Oi (included in maximum optimization) directs the compiler to use these intrinsic forms for all occurrences of the relevant functions. It is probably safer to enforce discipline through the **intrinsic** pragma whenever one of these standard functions is used in a resident module.

Building the Libraries

That completes the third component of the scheme this book will employ for writing resident programs in C. The essential (and most intricate) part of the scheme is the CRTKEEPC library which provides the segmentation model and rearranges segments in memory so that all resident material lies at the base of the program. As has been stated the segmentation model may be extended by assembling an edited KEEPSEGS.INC file (which would in fact be supplied in the form of an assembly language source file KEEPSEGS.ASM) and placing the resulting object module first in the chain of files to be linked. Even in this case the CRTKEEPC library must be linked before any other module that specifies an entry point for the executable. In general however, the library must be linked first.

The third part of the scheme—the KEEPCLIB library of support routines—needs no direct attention at link time because of the **INCLUDELIB** statement in KEEPINIT.ASM, but for compilation, programmers may want to use the definitions and function prototypes in the following header file:

```
/* KEEPC.H */

/*  Definitions and function prototypes for use with the KEEPC library  */

/*  A pseudo-keyword _RESIDENT is provided to help with references to
    resident data. */

#ifndef _RESIDENT_DEFINED
#ifndef _KEEP_
static _segment spKEEP = _segname ("KEEP");
#define _RESIDENT _based (spKEEP)
#else
#define _RESIDENT
#endif
#define _RESIDENT_DEFINED
#endif

/*  The default break address - held in DGROUP and therefore unavailable
```

```
    to the resident part of a program: */

extern void _far *lpBreakAddress;

/*  Functions declared in the resident code segment KEEP_TEXT  */

void _far _cdecl ChainIntr (void (_cdecl _interrupt _far *) ());

unsigned long _cdecl shl (unsigned long, unsigned);
unsigned long _cdecl shr (unsigned long, unsigned);

/*  Functions declared in the non-resident code segment _TEXT  */

typedef void (_cdecl _far *XKEEPC) (void);

int _cdecl AtExitKeepC (XKEEPC);
void _cdecl CloseAllFiles (void);
void _cdecl ExitKeepC (unsigned, void _far *);
unsigned _cdecl FreeEnvironment (void);
XKEEPC _cdecl OnExitKeepC (XKEEPC);
```

Management of the three distributable components is eased by a makefile prepared to run under the NMAKE utility included with version 6.0 and higher of both MASM and the Microsoft C compiler:

```
#   This makefile produces the libraries and object modules for the
#   Keep C scheme for writing resident C programs capable of discarding
#   code and data not needed after initialization.

!INCLUDE <standard.mak>

TARGETS : keepsegs.obj crtkeepc.lib keepclib.lib

#   KEEPSEGS.OBJ - source supplied to allow extension of the segmentation
#   model. If used, it must be the first module to be linked.

keepsegs.obj : keepsegs.asm {$(INCLUDE)}keepsegs.inc

#   CRTKEEPC.LIB - the Keep C interface with the C run-time. This should be
#   placed first in the list of modules to be linked, with the one noted
#   exception.

KEEPC_INCLUDES = keepinit.inc {$(INCLUDE)}keepsegs.inc
```

```
!IFNDEF SEGDEBUG
SEGDEBUG = ON
!ENDIF
!IF "$(SEGDEBUG)" == "ON"
SEGDEBUG_OBJ = segdebug.obj
SEGDEBUG_DEF = /D _SEGDEBUG_
!ENDIF

keepinit.obj : keepinit.asm $(KEEPC_INCLUDES)

progname.obj : progname.asm $(KEEPC_INCLUDES)

!IF "$(SEGDEBUG)" == "ON"
segdebug.obj : segdebug.asm $(KEEPC_INCLUDES)
!ENDIF

swaprefs.obj : swaprefs.asm $(KEEPC_INCLUDES)
  $(ASSEMBLE) $(SEGDEBUG_DEF) swaprefs.asm

swapsegs.obj : swapsegs.asm $(KEEPC_INCLUDES)

crtkeepc.lib : keepinit.obj progname.obj $(SEGDEBUG_OBJ) swaprefs.obj \
               swapsegs.obj
  !$(LIBRARIAN) crtkeepc.lib -+$?;
  del crtkeepc.bak

#   KEEPCLIB.LIB - optional support for resident programming.

chainint.obj : chainint.asm

closeall.obj : closeall.c

free_env.obj : free_env.c

keepexit.obj : keepexit.asm

onexit.obj : onexit.asm

shift.obj : shift.c
  $(COMPILE) /Gs /ND KEEP /NT KEEP_TEXT shift.c

keepclib.lib : chainint.obj closeall.obj free_env.obj keepexit.obj \
               onexit.obj shift.obj
  !$(LIBRARIAN) keepclib.lib -+$?;
  del keepclib.bak
```

An Example—Importing Material for Residence

Consider a fragment that require no resident coding from the programmer but which shows how library material (such as the programmer may have written for another purpose) may be incorporated effortlessly into the Keep C scheme. The floating-point emulator provided with Microsoft's high-level languages is certainly suitable for incorporation. Although it requires fairly substantial support routines in the _TEXT segment and accompanying data in DGROUP, the emulator itself is accessed via software interrupt and lies entirely within the special segments EMULATOR_TEXT and EMULATOR_DATA. Thus, the following KEEPSEGS.ASM file, which is merely the KEEPSEGS.INC file with the comments stripped and the two library segments added, will ensure correct placement of the segments within the scheme for swapping resident data with non-resident code.

```
; KEEPSEGS.ASM

;   Keep C segmentation model adapted for retention of the Microsoft
;   floating-point emulation library.

KEEP_BTEXT                  SEGMENT PARA COMMON 'CODE'
KEEP_BTEXT                  ENDS
KEEP_TEXT                   SEGMENT WORD PUBLIC 'CODE'
KEEP_TEXT                   ENDS
EMULATOR_TEXT               SEGMENT PARA PUBLIC 'CODE'
EMULATOR_TEXT               ENDS

NOKEEP_BTEXT                SEGMENT PARA COMMON 'CODE'
NOKEEP_BTEXT                ENDS

KEEP_BDATA                  SEGMENT PARA COMMON 'FAR_DATA'
KEEP_BDATA                  ENDS
KEEP                        SEGMENT WORD PUBLIC 'FAR_DATA'
KEEP                        ENDS
KEEP_CONST                  SEGMENT WORD PUBLIC 'FAR_DATA'
KEEP_CONST                  ENDS
KEEP_BSS                    SEGMENT WORD PUBLIC 'FAR_DATA'
KEEP_BSS                    ENDS
KEEP_GROUP                  GROUP    KEEP, KEEP_CONST, KEEP_BSS
EMULATOR_DATA               SEGMENT PARA PUBLIC 'FAR_DATA'
EMULATOR_DATA               ENDS

NOKEEP_BDATA                SEGMENT PARA COMMON 'FAR_DATA'
NOKEEP_BDATA                ENDS

END
```

Keeping the emulator in memory is an almost trivial task now that attention need not be paid to the messy details of segment arrangement. Ordinarily, the emulator is both installed and removed by the program with which it has been linked. All that need be done here is to persuade the program's C terminator that the emulator was never part of the program. All the interrupts used for entry to the emulator will be left hooked and the Keep C library will ensure that the emulator's memory is retained. Remember to link KEEPSEGS.OBJ + CRTKEEPC.LIB first.

```
/* FP.C */

#include    <dos.h>
#include    <signal.h>

#include    <keepc.h>

/*  The variable _fpinit contains the address at which to call the emulator
    for control - clear this to pretend that the emulator was never present,
    so it won't be removed at termination. Note that robust adaptation of the
    emulator requires knowledge and use of the interface at this address. */

extern void _far * _fpinit;

extern int _fltused; // external reference to ensure that
                     //   the emulation library is linked

static int _far HardErrorTrap (void)  // fail any critical errors
{                                     //   while closing file handles
    return (_HARDERR_FAIL);
}

void main (void)
{
    /*   Make sure the external reference to _fltused actually appears in the
         object file. An alternative is to operate non-trivially on a
         variable of type float, then the compiler will generate the external
         reference itself. */

    int dummy = _fltused;

    signal (SIGINT, SIG_IGN);         // standard provisions to prevent
    _harderr (HardErrorTrap);         //   premature termination

    fpinit = 0;                       // clear _fpinit, as described above
    CloseAllFiles ();                   // tidy up
    FreeEnvironment ();
    ExitKeepC (0, 0);                   // keep the emulator's memory
}
```

Readers may suspect that some drastic simplification has taken place, that notably, the emulator's initialization involves the specification of an address for handling exceptions and that this will be invalidated when the program terminates. The point, however, is not to rework floating-point emulation, only to put the scheme developed in this chapter to a test of flexibility. If the Keep C scheme is versatile enough to handle the retention even of foreign material (and the floating-point emulator is just a convenient example of library code with its own segmentation), then many of the details of internal structure in resident programming can truly be forgotten.

Readers who want to experiment with floating-point emulation may want to know the values used by the linker to generate emulated 80x87 instructions. Assembling or compiling with the in-line emulator option /FPi produces real 80x87 code but with WAIT instructions and fixup records. This is why a genuine coprocessor library may be specified at link time. For such a case, the fixups are all zero, as they are in the module RMFIXUPS.OBJ, which is normally in the compiler's LIB directory (along with a few other small object files for specific purposes). If the fixups aren't supplied explicitly, the compiler finds them in the standard library. For the emulator versions of the library, the fixups are the following, but with additional references which draw in all the emulator's code and data.

```
; FPFIXUPS.ASM

;   Public constants required by the linker to convert WAIT instructions to
;   interrupts (34h to 3Dh inclusive) for the floating-point emulator:

@fpfixup        MACRO   label, value
                        PUBLIC  label
label                   =       value
                        ENDM

@fpfixup        FIARQQ, 0FE32h          ; first fixup for WAIT DS: ESC
@fpfixup        FICRQQ, 0e32h           ; first fixup for WAIT CS: ESC
@fpfixup        FIDRQQ, 5c32h           ; fixup for WAIT ESC
@fpfixup        FIERQQ, 1632h           ; first fixup for WAIT ES: ESC
@fpfixup        FISRQQ, 0632h           ; first fixup for WAIT SS: ESC
@fpfixup        FIWRQQ, 0A23dh          ; fixup for NOP WAIT

@fpfixup        FJARQQ, 4000h           ; second fixup for WAIT DS: ESC
@fpfixup        FJCRQQ, 0C000h          ; second fixup for WAIT CS: ESC
```

```
@fpfixup            FJSRQQ, 8000h             ; second fixup for WAIT SS: ESC

END
```

By supplying these same fixups on their own the emulator will not be linked, but the 80x87 instructions (with the WAIT prefixes) are converted to interrupts, just the same—and are just right for invoking an emulator already present in memory.

Another Example—Writing a Resident Program

Now let's turn to an example in which the programmer must provide the resident material rather than import it ready-made. The first task, especially given the enforced separation of resident and non-resident modules, is to isolate the program's permanent needs. Looking ahead to the forthcoming exploration of extended memory in Part 3, the example chosen here is a program that "eats" an amount of extended memory.

Modification of the PC architecture to support the 80286 processor for the PC-AT was reflected by the provision of a host of new BIOS services on int 15h. One of these, function 88h, returns with register ax containing the amount of extended memory installed on the computer. By trapping other programs' attempts to ask the BIOS how much extended memory exists and by reporting a smaller amount than really exists, the example program (called INT15MEM) usurps the difference. When INT15MEM is first run, the value it obtains by calling int 15h need not reflect the amount of extended memory physically present—other programs may have already played the same game, and, indeed, this deception is a time-honoured technique for obtaining extended memory. The problem with this method is that allocation is necessarily from the top, so memory cannot be obtained from and returned to the system in arbitrary order. At a low enough level, however, there is simply no workable alternative, and use of extended memory on the PC depends on programs like HIMEM.SYS which obtain memory this way only to break it up and manage it for general convenience.

The resident module of INT15MEM needs the address of the int 15h handler to which it must pass requests that do not have AH = 88h, and it needs a variable containing the amount of extended memory it will claim is present. Add the int 15h handler itself and the resident module is complete:

```
/* HANDLER.C */

#include     <intpack.h>
#include     <keepc.h>

/*  The int 15h handler intercepts requests for the amount of available
    extended memory, returning a smaller amount and in effect allocating
    the remainder for its own purposes (in this case, for nothing). */
```

```
void (_interrupt _far *lpOrgInt15h) (void) = 0;
unsigned MemLeftInKB = 0;

void _interrupt _far int15h (struct INTPACK r)
{
    if (r.ah == 0x88) r.ax = MemLeftInKB;
    else ChainIntr (lpOrgInt15h);
}
```

Nothing else *need* be kept in memory and nothing else *will* be kept.

As a resident module, HANDLER.C *must* be compiled with the switches /ND KEEP /NT KEEP_TEXT to ensure that the code and data appear in the segments expected by the Keep C scheme. Ordinarily the /Gs switch would also be needed to prevent the compiler from generating a call to `__aNchkstk` or some similar library function for checking that the stack is not about to overwrite near data, but this simple example contains only an **_interrupt** function, for which the compiler does not already insert stack probes.

The non-resident part of the program, which follows, is more or less self-explanatory, except perhaps for the `_RESIDENT` keyword (in fact a macro defined in the KEEPC.H header file). This could in fact be replaced by **_far** but serves as a reminder that the data concerned lies in a resident segment specifically. There is also a technical advantage arising because the `_RESIDENT` pseudo-keyword defines a variable `spKEEP` (private to each file in which it is used) and initializes it with the segment address of the resident data group. Thus, using `_RESIDENT` reduces the number of relocation entries generated by references to resident data—the sort of optimization one might expect from the Microsoft C **same_seg** pragma (but which does not appear to work).

```
/* MAIN.C */

#include    <dos.h>
#include    <signal.h>
#include    <stdarg.h>
#include    <stdio.h>
#include    <stdlib.h>

#include    <keepc.h>
#include    <standard.h>

static char description [] = "\n\
Allocates extended memory through the int 15h interface.\n\
\n\
  INT15MEM [kilobytes]\n\
\n\
Without an argument, the program reports the current availability.\n";

/*  External references to the resident int 15h handler and two resident
    data variables - all declared in HANDLER.C  */

extern void _interrupt _far int15h (void);

extern void (_interrupt _far * _RESIDENT lpOrgInt15h) (void);
extern WORD _RESIDENT MemLeftInKB;

/*  The determination of memory available through the int 15h interface is
    set aside for a separate function, since it uses in-line assembler. */

static WORD GetExtMemInKB (void);

/*  For safety while terminating and staying resident  */
static int _far HardErrorTrap (void)
{
    return (_HARDERR_FAIL);
}

/*  Message output  */

static void PutError (char *str, ...);

/*  ====================================================================  */

int main (int argc, char **argv)
{
    register char *ptr;
    WORD AvailMemInKB;
    register WORD MemToAllocateInKB = 0;

    /*    Parse the command line. */
    while (++ argv, -- argc) {
          ptr = *argv;
          if (*ptr == '/') {
              if (ptr [1] == '?' AND ptr [2] == '\0') {
                    printf (description);
                    return (0);
              }
              PutError ("invalid switch %s", ptr);
              return (-1);
```

```
        }
        else {
            if (MemToAllocateInKB != 0) {
                PutError ("too many parameters %s", ptr);
                return (-1);
            }
            MemToAllocateInKB = atoi (ptr);
            if (MemToAllocateInKB == 0) {
                PutError ("non-zero number expected, not %s", ptr);
                return (-1);
            }
        }
    }

    /*  Determine the amount of extended memory currently available and
        report it. If no allocation is also desired, then exit. Otherwise,
        check that the allocation does not exceed the amount available. */

    AvailMemInKB = GetExtMemInKB ();
    printf ("\nAmount of extended memory available through int 15h is %dKB\n",
        AvailMemInKB);

    if (MemToAllocateInKB == 0) return (0);

    if (MemToAllocateInKB > AvailMemInKB) {
        PutError ("insufficient extended memory available");
        return (-1);
    }

    /*  Everything's in order. Now work out how much memory will be left
        behind and hook int 15h so that all future requests for extended
        memory will think that the reduced amount is all that exists. Do
        not forget to prevent DOS exceptions from terminating the program
        while the int 15h vector is hooked. */

    signal (SIGINT, SIG_IGN);
    _harderr (HardErrorTrap);

    printf ("Now allocating %dKB of extended memory, ", MemToAllocateInKB);
    MemLeftInKB = AvailMemInKB - MemToAllocateInKB;
    printf ("leaving %dKB behind.\n", MemLeftInKB);

    lpOrgInt15h = _dos_getvect (0x15);
    _dos_setvect (0x15, int15h);
    /*  Tidy up. */
    CloseAllFiles ();
    FreeEnvironment ();

    /*  Use the ExitKeepC function to terminate, but note that in this
        simple case, _dos_keep (0, (_segment) lpBreakAddress - _psp) is
        equivalent. */

    ExitKeepC (0, 0);
}

static WORD GetExtMemInKB (void)
{
    _asm {
            mov     ah,88h
            clc
            int     15h
            jnc     done
            xor     ax,ax
        done:
    }

}

static void PutError (char *str, ...)
{
    va_list va_start (argptr, str);
    printf ("\nINT15MEM error: ");
    vprintf (str, argptr);
    printf ("\n");
    va_end (argptr);
}
```

To help ensure that resident modules be compiled with the necessary switches, it is recommended that projects be managed with makefiles which include

```
# KEEPC.MAK

KEEP = /D _KEEP_ /Gs /ND KEEP /NT KEEP_TEXT
```

or at least the line it contains. With this included, resident modules may be compiled by specifying the command `$(COMPILE) $(KEEP)` in their description blocks. A makefile for the INT15MEM program would therefore look like this:

```
!INCLUDE <standard.mak>
!INCLUDE <keepc.mak>

TARGETS : int15mem.exe

handler.obj : handler.c
  $(COMPILE) $(KEEP) handler.c

main.obj : main.c

int15mem.exe : handler.obj main.obj
  $(LINK) crtkeepc.lib+handler+main,int15mem.exe;
```

Note that the INT15MEM program requires hardly more effort than a toy or gadget ought to and that despite having the convenience of the `printf` function during installation, the program's memory footprint is only 384 bytes (including a more or less unavoidable 256 for the PSP).

To run INT15MEM and not be told that no extended memory is available requires one of the following:

- that HIMEM.SYS is not installed (nor any other XMS server);
- that no program that uses HIMEM's extended memory management services has been loaded;
- that HIMEM has been directed through its **/int15** switch to leave some memory available for programs to obtain via int 15h.

Many readers will have a write-protected system disk containing a minimal CONFIG.SYS and possibly also containing a host of their favorite system-level utilities (those who haven't ought to, especially in these days when even professional programmers may unwittingly infect machines with computer viruses); after rebooting with such a disk, INT15MEM may be run repeatedly to set aside all the machine's extended memory. What would be more useful, given the importance of HIMEM.SYS to extended memory management, would be to run the program before and after HIMEM's installation, that is, to have this same INT15MEM program also load as a device driver.

There remains a small problem with the Keep C library: A resident program that contains no resident code material confuses the KEEP_BTEXT and NOKEEP_BTEXT fences. The easy way to fix this is to force a dummy paragraph of resident code.

CHAPTER 7

Writing Device Drivers as C Programs

The archetypical device driver is essentially an overlay for the DOS kernel. It is a piece of program code that DOS loads into memory to gain some additional functionality such as to support extra hardware. The device driver is a run-time extension of the operating system rather than a program in its own right. This special relationship is reflected in a structural design to which device drivers must conform, but which is not easy to accommodate when writing in a high-level language.

The Essential Components

An obvious problem is that the device driver must begin with a small amount of data—the device driver header, which describes the driver and locates its two Strategy and Interrupt calling points. As noted in Chapter 6 most high-level language implementations employ some mechanism to ensure that programs begin with code. This means that the device driver header must be the first object declared in the first code segment. The facility offered by many compilers (including Microsoft's) to emit data in code segments is not intended to cope even with simple structures and is in any case unlikely to allow emission outside of functions, so, unavoidably, the device driver header must be supplied in an assembly language module.

Writing the strategy and interrupt routines in C is feasible only if the means exist to preserve all the caller's registers. For the strategy routine some ability to read an argument passed in registers is also needed. Although DOS and the driver exchange data in a special structure known as a request packet, the address of the structure is passed to the strategy routine in the register pair es:bx. Preservation of and access to the caller's registers is easy enough in Microsoft C for functions declared with the `_interrupt` keyword, because the compiler generates prologue code to save the registers in a stack frame which may then be addressed through the function's formal parameter list. Epilogue code pops the registers from the stack before issuing an `IRET` instruction. By default the caller's registers are preserved by the handler, but, if necessary, values may be returned in registers by modifying the stacked copies before closing the function. The fragment

```
#include    <driver.h>                  /*  for the REQUEST_HEADER structure   */
#include    <intpack.h>                 /*  for the INTPACK structure */
```

```
#include    <standard.h>    /*  for the MK_FP macro */

struct REQUEST_HEADER _far *lpRequestHeader;

void _interrupt _far handler (struct INTPACK r)
{
    lpRequestHeader = MK_FP (r.es, r.bx);
            .
            .
            .
}
```

shows how easily an interrupt handler written in C could retrieve the far pointer passed in es:bx and preserve the caller's registers.

The strategy and interrupt routines are far procedures, not interrupt handlers, but the same prologue code is generated in response to the combination of `_saveregs` and `_loadds` keywords. The stacked registers cannot be addressed so easily, and the epilogue code is designed to return a value by default rather than to preserve all registers—even if the function returns type `void`. A strategy routine that begins by saving the address of the request header must therefore resemble this:

```
#include            <driver.h>
#include            <intpack.h>

struct REQUEST_HEADER _far *lpRequestHeader;

static struct INTPACK intpack;     /* a necessary ruse to access structure
                                   elements inside an _asm block if
                                   names are non-unique */

void _far _saveregs _loadds StrategyRoutine (void)
{
    _asm {
            mov         bx,[bp]intpack.bx
            mov         [lpRequestHeader],bx
            mov         es,[bp]intpack.es
            mov         [lpRequestHeader + 2],es
    }
            .
            .
            .

    _asm    mov         ax,[bp]intpack.ax
}
```

In-line assembler is needed at the start to retrieve es:bx and at the end to counteract the epilogue code (which will copy ax back to the stack). Comparison with the earlier fragment is hardly favorable, but the main point is that it is possible to write a DOS device driver with a modern C compiler, save only for the device-driver header.

C Functions for Driver Commands

A typical device-driver interrupt routine examines the command number in the request packet to determine which service the DOS kernel requires and then branches to some relevant code. This suggests an interface in which the programmer provides C functions for each of the device driver commands. Since all data exchanged between DOS and the driver passes through the request packet, the functions need be called with only one argument—a far pointer to the request structure. For all device driver commands (except initialization) DOS expects the driver to place a status in the appropriate request header field, and since some commands need provide no other information, it simplifies matters to have all the C-language device driver functions return the status in the normal way so that the interrupt routine may enter it in the request header on their behalf. A scheme emerges in which the programmer need only be concerned with high-level language entry points for the various device driver commands, all conforming to the prototype

```
unsigned _near _cdecl DrvrProc (struct REQUEST_HEADER _far *);
```

and with names that are descriptive of the corresponding commands:

```
DrvrProcInit
DrvrProcMediaCheck
DrvrProcBuildBPB
DrvrProcIOCTLRead
DrvrProcRead
DrvrProcNondestructiveRead
DrvrProcInputStatus
DrvrProcInputFlush
DrvrProcWrite
DrvrProcWriteWithVerify
DrvrProcOutputStatus
DrvrProcOutputFlush
DrvrProcIOCTLWrite
DrvrProcOpenDevice
DrvrProcCloseDevice
DrvrProcRemovableMedia
DrvrProcOutputUntilBusy
DrvrProcGenericIOCTL
DrvrProcGetLogicalDevice
DrvrProcSetLogicalDevice
DrvrProcIOCTLQuery
```

Transfer of control from the interrupt routine to the right high-level entry point is implemented in C by using an array of function pointers, as shown in the following code outline (to be considered as an extension of the earlier fragment that contains the strategy routine). A function `DrvrProcBadCommand` is introduced to deal with command numbers that lie out of bounds by returning an appropriate error.

```
extern unsigned DrvrProcInit (struct REQUEST_HEADER _far *);
extern unsigned DrvrProcMediaCheck (struct REQUEST_HEADER _far *);
          .
          .
          .

unsigned (_near *DrvrProcTable []) (struct REQUEST_HEADER _far *) = {
    DrvrProcInit,
    DrvrProcMediaCheck,
          .
          .
          .

};

unsigned _near DrvrProcBadCommand (struct REQUEST_HEADER _far *lp)
{
    return (D_ERROR | D_DONE | D_BAD_COMMAND);
}

void _far _saveregs _loadds InterruptRoutine (void)
{
    register unsigned (_near *funcptr) (struct REQUEST_HEADER _far *);
    unsigned char command = lpRequestHeader -> cCommand;

    if (command < sizeof (DrvrProcTable) / sizeof (DrvrProcTable [0]))
         funcptr = DrvrProcTable [command];
    else funcptr = DrvrProcBadCommand;

    lpRequestHeader -> wStatus = funcptr (lpRequestHeader);

    _asm    mov     ax,[bp]intpack.ax
}
```

Unknown Commands

For almost all device drivers, only a few of the functions described in the array will play a non-trivial role. This is a compelling reason to construct the array of function pointers in assembly language, where the `EXTERN` directive may allow the specification of an alternative name with which the linker may resolve the external reference if the target has not been supplied in another module. Thus the programmer need be concerned only with the functions that have some relevance to the device driver at hand. All functions omitted will be defaulted by the linker to one that returns the "unknown command" error.

It may be worth a small digression to justify this choice of default behavior, given that the built-in NUL device driver, whose inertness is its claim to existence, responds to all commands by simply setting the DONE bit in the request header's status field. Many common device drivers mimic this reaction for commands they do not handle non-trivially. The practice is not adopted here for reasons the reader may appreciate after trying any of the (more or less) normal DOS commands

```
type emmxxxx0 >tempfile
type ms$mouse >tempfile
type xmsxxxx0 >tempfile
```

when the relevant device driver (EMM386.EXE, MOUSE.SYS, and HIMEM.SYS, respectively) is loaded. None of these devices are intended to provide I/O through the device driver interface. If not interrupted by something like Ctrl-C, these commands will continue until the available disk space has been exhausted. Setting the DONE bit makes it seem as if requests for more input are always met. The NUL device driver escapes this problem because when DOS is asked via functions 3Fh or 40h to read from or write to a character device driver that has the NUL bit set in its attribute word, the request is simply not passed to the driver. That this is not so for the obsolete functions numbered 01h to 0Ch may be seen by compiling

```
/*  test.c  */
#include    <conio.h>

void main (void)
{
    unsigned char ch;
    do {
          ch = getche ();     // essentially int 21h function 01h
    } while (ch != 0x1A);
}
```

and then running

```
test <nul >tempfile
```

to achieve the earlier effect and invalidate the notion that the NUL device provides no input.

Thus, by having the device driver return an error instead of setting the DONE bit, DOS will be forced (in most cases) to generate an int 24h exception. Even though the older DOS functions have little ability to report errors directly to applications, at least some indication of the problem may be sent to the application and probably even to the user through some message like

```
Invalid device request reading device SMARTAAR
Abort, Retry, Ignore, Fail?
```

For many driver commands the question of default behavior may be avoided because the command will never be called unless a corresponding bit flag is set in the device driver's attribute word. Note finally that the question is completely irrelevant (at DOS function level) for a character device driver whose logical name consists only of spaces (for instance), since the device can never be opened and only the initialization command will ever be processed.

The alternative name feature in the assembly language `EXTERN` directive also allows extension of the strategy routine to do more than just save the request header's address, but without requiring any attention whatsoever from the programmer should the facility not be wanted. The occasions on which this generalization will be useful must be very few. One example is offered by Microsoft's own EMM386.EXE, which has quite a large strategy routine to unpack the remainder of the driver, if necessary, before the interrupt routine proceeds with the initialization.

A Front End

Thus, despite the feasibility in C of everything but the device driver header, a definite advantage in the context of providing a general framework for writing device drivers in C can be gained by placing all the basic driver components—the header, plus skeletal strategy and interrupt routines—in a single assembly language source file, as follows. For some purposes, most notably when the file is to contain more than one driver or when unsupported commands must be treated differently (perhaps by chaining the request to a previously-loaded driver), it may be necessary or desirable to redesign DRIVER.ASM slightly. The resulting object file DRIVER.OBJ must be linked first to ensure that the device-driver header will indeed lie at the beginning of the driver. The file adopts the Keep C segmentation model from the previous chapter to help arrange for initialization code to be discarded once the device driver has been safely installed.

```
; DRIVER.ASM

.NOLIST
  INCLUDE          driver.inc
  INCLUDE         keepsegs.inc
.LIST

.MODEL                      SMALL

;   The device driver's attribute bits and name may be supplied as symbols
;   at assembly time. Otherwise, they take default values which could be
;   modified by a utility program after assembly and linkage—this would
;   permit distribution as an object module. The default values are
;   suitable for programs cast as character device drivers merely to get
;   code into memory as early as possible during system startup.

IFNDEF                      _ATTRIBUTE_
  _ATTRIBUTE_               EQU      IS_CHAR_DEVICE
ENDIF

IFNDEF                      _NAME_
  _NAME_          TEXTEQU <"">
ELSE
  _NAME_          CATSTR   <!">, _NAME_, <!">
ENDIF

;   ====================================================================

.CODE                       KEEP_TEXT

;   The device-driver header this must be located at the very beginning of
;   the device driver:

                            PUBLIC   C DriverHeader
DriverHeader                DRIVER_HEADER      { \
                                               0FFFFFFFFh,
                                               _ATTRIBUTE_,
                                               OFFSET KEEP_TEXT:strategy,
                                               OFFSET KEEP_TEXT:interrupt,
                                               {_NAME_} \
                                               }

;   ====================================================================

.FARDATA          KEEP

;   Jump table for device driver commands:

                            PUBLIC   C DrvrProcTable
```

```
DrvrProcTable                LABEL    WORD

;   All procedures in the table have C language prototypes of the form

;         unsigned _near _cdecl DrvrProc (struct REQUEST_HEADER _far *)

;   Functions not provided in other modules default to one which returns the
;   unknown command error, except that initialization is compulsory.

NEXTNUM            =         00h
@DrvrProc          MACRO     num:REQ, command:REQ
  IF                         num LT NEXTNUM
    .ERR           <@DrvrProc macro not in sequence>
  ENDIF
  IF                         num GT NEXTNUM
    REPEAT                   num - NEXTNUM
                             dw       OFFSET KEEP_TEXT:DrvrProcBadCommand
    ENDM
  ENDIF
  .CODE            KEEP_TEXT
  IF                         num EQ 00h
                             EXTERN   DrvrProc&command:NEAR
  ELSE
                             EXTERN   DrvrProc&command (DrvrProcBadCommand):NEAR
  ENDIF
  .FARDATA                   KEEP
                             dw       OFFSET KEEP_TEXT:DrvrProc&command
  NEXTNUM          =         num + 01h
ENDM

@DrvrProc          00h, Init
@DrvrProc          01h, MediaCheck
@DrvrProc          02h, BuildBPB
@DrvrProc          03h, IOCTLRead
@DrvrProc          04h, Read
@DrvrProc          05h, NondestructiveRead
@DrvrProc          06h, InputStatus
@DrvrProc          07h, InputFlush
@DrvrProc          08h, Write
@DrvrProc          09h, WriteWithVerify
@DrvrProc          0Ah, OutputStatus
@DrvrProc          0Bh, OutputFlush
@DrvrProc          0Ch, IOCTLWrite
@DrvrProc          0Dh, OpenDevice
@DrvrProc          0Eh, CloseDevice
```

```
@DrvrProc          0Fh, RemovableMedia
@DrvrProc          10h, OutputUntilBusy
@DrvrProc          13h, GenericIOCTL
@DrvrProc          17h, GetLogicalDevice
@DrvrProc          18h, SetLogicalDevice
@DrvrProc          19h, IOCTLQuery

;   ------------------------------------------------------------------------

;   Storage variable for the request header's address:

                            PUBLIC   lpRequestHeader
lpRequestHeader   dd        00000000h

;   ========================================================================

.CODE                       KEEP_TEXT

strategy           PROC     FAR PRIVATE USES ds ax

;   Save the address of the request header.

                            mov      ax,KEEP_GROUP
                            mov      ds,ax
                            ASSUME   ds:KEEP_GROUP

                            mov      word ptr [lpRequestHeader],bx
                            mov      word ptr [lpRequestHeader + 02h],es

;   Provide the means to do more than save the request header's address.
;   This will be needed only very rarely, so leave the responsibility for
;   preserving registers to the DrvrProcStrategy procedure, a suitable
;   prototype for which is:

;   void _near _saveregs _cdecl DrvrProcStrategy (struct REQUEST_HEADER _far *)

;   Direct the linker to resolve the external reference with a simple near
;   return if no other module provides the DrvrProcStrategy function.

                            EXTERN   DrvrProcStrategy (DrvrProcRetn):NEAR

                            push     es
                            push     bx
                            call     DrvrProcStrategy
                            add      sp,04h
```

```
                        ret

strategy        ENDP

;   ----------------------------------------------------------------

interrupt       PROC    FAR PRIVATE USES es ds bp di si dx cx bx ax

                        mov     ax,KEEP_GROUP
                        mov     ds,ax
                        ASSUME  ds:KEEP_GROUP

;   Determine which procedure from the jump table (if any) will handle the
;   command.

                        les     bx,[lpRequestHeader]
                        mov     al,es:(REQUEST_HEADER PTR [bx]).cCommand

                        cmp     al,NEXTNUM
                        jnb     too_high

                        xor     ah,ah
                        mov     si,ax
                        shl     si,1
                        mov     si,[si + DrvrProcTable]

                        jmp     @f
too_high:
                        mov     si,OFFSET DrvrProcBadCommand
@@:

;   Call the relevant C function, passing the address of the request header.
;   Note that the procedure will be called with ds already addressing the
;   resident data, although in practice functions written in C will probably
;   load ds explicitly.

                        push    es
                        push    bx
                        call    si
                        add     sp,04h

;   Each procedure is required to return a value for the device driver
;   status word.

                        les     bx,[lpRequestHeader]
                        mov     es:(REQUEST_HEADER PTR [bx]).wStatus,ax
                        ret

interrupt       ENDP

;   -----------------------------------------------------------------

DrvrProcBadCommand PROC     NEAR PUBLIC

                        mov     ax,D_ERROR OR D_DONE OR D_BAD_COMMAND

DrvrProcRetn            PROC    NEAR PUBLIC

                        ret

DrvrProcRetn            ENDP
DrvrProcBadCommand ENDP

END
```

Initialization in C

After initialization, code in the device driver is executed by far calls from the DOS kernel as part of its int 21h functionality or from a program such as PRINT.EXE that cooperates very closely with the kernel. When DOS is this deep within its processing it is certainly not prepared for another int 21h request. In the presence of Windows or a network redirector, it shrouds all its device driver calls with int 2Ah critical section signals to announce that reentry would be unwelcome. Device drivers augment int 21h services; they are not able to request them.

Using DOS Functions

During device driver initialization the caller is not the DOS kernel but the SYSINIT module which, having initialized the MSDOS.SYS file and constructed a minimal set of necessary data structures, may access all the int 21h services the kernel can offer. The file system is operational, albeit in a rudimentary stage of development, and after consulting the CONFIG.SYS text file, SYSINIT proceeds to load device drivers into memory for initialization. It is often said that only a few int 21h functions are available for use during device driver initialization, but given that DOS is clearly able to perform file I/O at this stage, readers will no doubt see such statements for what they are.

Because certain system resources are prepared only minimally, some DOS functions may fail unexpectedly. Software that hopes to be considered well-written even under normal conditions will check all DOS calls for error conditions and will not be troubled by the fact that SYSINIT has only three file handles open by default (all for the default CON device, at least initially) or that DOS has only five System File Tables (SFTs). Access to the AUX and PRN devices should not be taken for granted, say by using DOS functions 03h to 05h or otherwise assuming that handles for stdaux and stdprn are open. As a defensive point, software should not rely on having five handles open by default.

Catastrophe—that is, a system crash rather than the simple return of an error condition—lurks within only one class of responsible int 21h usage during device-driver initialization. At this stage, DOS has no SFTs dedicated to File Control Block (FCB) operations, and since the DOS kernel does not check for this condition before it begins searching for a suitable SFT, all FCB functions that end up using an SFT must be considered off-limits. The functions concerned are the ones that open, read, write, and close, but not those that find, delete, or rename. DOS attempts to close all FCB-SFTs (again assuming that at least one exists) as part of its normal clean-up operations during process termination.

Using a Run-time Library

The environment in which device drivers are initialized is quite similar to that experienced by normal programs except that device driver code is called as a far procedure, not executed as a separate process. From the perspective of high-level programming, this places off-limits the large number of useful functions in the standard run-time library that assume normal program conditions either through their reliance on data recorded by startup code or more subtly through their treatment of certain error conditions as fatal. The latter is particularly important because the active process during device driver initialization is the SYSINIT module, which uses only a temporary PSP that is wholly unprepared for termination.

If device drivers are to be allowed full use of high-level programming facilities for their initialization, they must be isolated from SYSINIT by having PSPs of their own so that they behave more like application programs. This may appear to be a substantial change to the normal path of system startup, but bear in mind that, ultimately, SYSINIT runs the **shell** as a child process and before then runs any programs loaded with the **install** directive, so constructing a separate PSP to handle device driver initialization is less a conceptual leap than a small step taken a little earlier than DOS's designers seem to have intended.

The usual means of executing a child is to call DOS function 4B00h. Apart from loading the program from disk and handling the details of memory allocation, this function creates a PSP containing (among other things) the segment address of the parent's PSP and the address at which that parent will resume control when the child has finished, whether voluntarily by calling a DOS termination function such as 31h or 4Ch, or less willingly as the result of a critical error. Ordinarily the child leaves its PSP alone, but the **shell** is an exception, since it must establish itself as its own parent and ensure that termination simply restarts the **shell**'s own command processing loop.

Device driver initialization is begun by a far call to the strategy and interrupt routines, not by invoking DOS function 4B00h. It is possible for these routines to emulate that DOS function's preparation of a child PSP and transfer of control—in effect, to reproduce the entry conditions normally applicable to programs. In the context of the front-end module DRIVER.ASM, this means providing a `DrvrProcInit` procedure that performs the emulation and then jumps to the program's normal entry point, as if it were starting a child process that happens to come from the same file. For a C program the run-time startup code will be executed as usual, and the programmer may regard the `main` function as the entry point for device-driver initialization (and still for normal program execution).

Transparency

Like the **shell**, the `DrvrProcInit` procedure must itself regain control on termination, not to prevent termination but to reproduce the normal exit conditions of a device driver so that SYSINIT will be unaware of the disruption. It is especially important to identify the data that a parent can retrieve from a child and to determine how much of the request packet can be filled without explicit action by the child. For as long as possible, the child should not *need* to know that it is initializing a device driver. The child may have been written as a stand alone program, designed for execution under a normal operating environment, but after relinking with DRIVER.OBJ and the CRTDRVR library that supplies the necessary `DrvrProcInit` function, this same program (whether it is intended to stay resident or not) should also be executable through a **device** statement in CONFIG.SYS, simply to have it run at the first opportunity during system startup.

The most noticeable piece of data communicated by a child process is the word the parent may obtain by calling DOS function 4Dh (Get Return Code). At C-language level, this word is the value returned by whichever member of the `spawn` family was used to run the child. The low byte is set explicitly by the child before it terminates and will be familiar to C programmers as the value they return in `main` or pass through the `exit` or `_dos_keep` functions. Under the control of the child this exit code cannot be used to glean information transparently. The high byte is recorded by the system to describe the method of termination, using the following values.

VALUE	TERMINATION METHOD	TERMINATION INTERFACE
00h	normal	int 20h, int 21h function 00h or int 21h function 4Ch
01h	Ctrl-C	int 23h exception
02h	critical error	int 24h exception
03h	resident	int 21h function 31h or int 27h

During device-driver initialization the DOS exceptions are defaulted by SYSINIT to handlers that restart any DOS function interrupted by Ctrl-C and fail any function that encounters a critical error. The child process may install handlers of its own, so all four termination methods are possible. The main distinction is between the first three and the last.

If the driver is not to stay in memory, perhaps because it is a real driver whose installation must be abandoned or because its residency was never intended, no information other than the termination method is needed. The `DrvrProcInit` procedure will be able to detect that the child component has not terminated via DOS function 31h and may then enter values in the request packet that are appropriate to aborting the installation. A standard request packet suffices. One variation is possible since the opportunity exists (introduced in DOS 4) to direct SYSINIT to display a message of the form

```
Error in CONFIG.SYS line 5
```

and it seems sensible to use the child's exit code to determine whether or not this message is desired. By convention, negative values indicate failure of some sort and will therefore be held to signal a desire for the system error message.

For a driver that intends to stay in memory SYSINIT needs more information, which the `DrvrProcInit` procedure must in turn be able to determine from the child component. Successfully initialized character device drivers need only indicate how much memory they want SYSINIT to allow for them. Block device drivers must also indicate the number of supported units and provide a BIOS Parameter Block (BPB) for each one. These extra pieces of data required for block device drivers play no role in process termination and will have to be communicated explicitly. The easiest way is to allow the child component to manipulate the request header directly, which has been made possible by declaring the far pointer `lpRequestHeader` in DRIVER.ASM to be **PUBLIC**. The memory requirement can be discerned transparently (for whichever type of driver), since the `DrvrProcInit` procedure, like any parent, will be able to examine the system-wide memory map to determine the amount and location of memory retained by its child after termination. For the situation at hand, matters are actually easier, since the `DrvrProcInit` procedure, having simulated the PSP (and in the implementation below, an accompanying memory arena header), will know exactly where to find the changes DOS makes as part of its bookkeeping for TSR termination.

Details of this bookkeeping could be left entirely within the annotation for the CRTDRVR.ASM source file were it not for some caution arising from a conflict between the PSP simulation and the meaning of the value passed in register dx to DOS function 31h or, equivalently, the second argument of the `_dos_keep` function. From DOS's perspective, the value is the number of paragraphs to which the memory block that begins with the program's PSP must be reduced. An algorithm derived directly from DOS's interpretation is to subtract the known segment address of the PSP from the lowest segment that lies above the PSP but is not required after termination. As an alternative, some programs add the total number of paragraphs of resident program material to a 16-paragraph allowance for the PSP. This method arrives at the same result except in one contrived case, but is technically less robust, because it relies on an observation about the load-time location of program material with respect to the PSP, which is one more than necessary to obtain the desired value. As a general rule, protection against change is inversely related to the number of unnecessary assumptions. Hopefully this good advice applied to this simple situation will not sound like special pleading after the revelation that the CRTDRVR library—which simulates a PSP low within the driver's memory image but not immediately preceding it—supports the transparent conversion of TSRs to load as device drivers if the permanent memory requirement is calculated by the first method but not the second.

Help From DOS 5

Under DOS 5 device drivers are given clear bounds to the area of memory available for use during initialization. This is an essential service now that drivers may be loaded into possibly small areas of upper memory, where an overrun as the driver builds data tables or tries to set aside buffer space may be all too likely. The problem is greatly diminished when loading device drivers into conventional memory but has been significant enough that DOS 5 performs some peculiar operations (see Chapter 3) to accommodate device drivers known to have sought an upper limit to available memory. New drivers may obtain the upper limit easily because SYSINIT places it in the request header—as a far pointer to the first byte of memory that the driver must not use—using the same field in which the driver is expected to return a far pointer to the first byte of memory the driver does not want.

For the CRTDRVR library, possessing a reliable upper bound is very important to ensure that the uninitialized data areas and stack (which are not part of the driver's image on disk and will not have figured in SYSINIT's attempt to avoid loading a driver into too little space) are in fact addressable, that the C run-time with which the library interfaces must be able to set up its internal memory management (for the near heap) correctly, and that the fake memory arena headers may seem just realistic enough that calls to DOS memory-management functions will behave well. Because this book centers on DOS 5 and higher (and, frankly, because no truly satisfactory solution exists in earlier DOS versions), no attempt is made to ascertain an upper limit to available memory if the DOS version is less than 5. Indeed, in the following form, the CRTDRVR library regards a lower DOS version as a fatal error condition.

The notion of DOS version has been complicated a little by the introduction of version deception in DOS 4 and now brought within the user's control via the SETVER program. A separate version number is maintained for each process so that a call to DOS function 30h may return some version other than the real one, depending on the name of the executable program. In DOS 5 this subterfuge is ordinarily not extended to overlays, which are not processes in their own right, but exceptions are made by default when SYSINIT is loading device drivers and can be forced at other times by using the new DOS function 4B05h (Set Execution State; see Chapter 5) to prepare the way. For present purposes, it is the real DOS version—obtainable by calling int 21h function 3306h—that is needed in order to establish that the request header contains an upper bound to memory.

Hindrance from Earlier Versions

Readers who embark on improving CRTDRVR.ASM to support earlier DOS versions may need to consider the more subtle question of critical error handling. Only since version 4 has SYSINIT dealt with critical errors by failing the DOS function that encountered the exception. Earlier versions used an int 24h handler more suited to those very early versions that supported only the ignore, retry, and abort options. This handler returns control to the DOS caller (SYSINIT) but does not set the carry flag to indicate the error. Instead, SYSINIT detects the occurrence of an error in some DOS functions by indirect inference. It sets the carry flag before issuing the int 21h instruction, knowing that successful completion of the DOS function will clear the flag. If the flag remains set when the function returns then an error has occurred.

In itself, this method is unsatisfactory, because some DOS functions call the kernel code for other functions. For instance, DOS function 4Bh calls code for opening a file, moving the file pointer, reading from the file, allocating memory, and so forth, each of which sets values to be returned in registers to the caller. This is why, until this day, the "load and execute" function corrupts registers ax, bx, and dx, whatever the claims of official literature. A success in an early stage may clear the carry flag (saved on the stack with the caller's other registers) and render SYSINIT ignorant of the later occurrence of a critical error (when the handler returns control directly to the DOS caller, restoring the flags from the stack).

As far as allowing device-driver initialization to use existing code such as that in a C run-time library, this method of critical error handling is fatal. Ordinarily, there is no need to set the carry flag before calling DOS functions. At the very least, support for earlier DOS functions requires the installation of a better int 24h handler after the activation (in the `DrvrProcInit` procedure) of the child PSP and before the transfer to the program's normal entry point.

The Run-time Interface Library

This completes the general observations leading to the motivation and feasibility of treating device driver initialization as normal program execution. For character device drivers the interface is essentially transparent. As with ordinary programs, the driver will be retained if a TSR exit is used, be it DOS function 31h or a high-level equivalent such as the `_dos_keep` function in the Microsoft C library. A normal exit or a fatal run-time error will cause the driver to be discarded as expected. Initialization of block device drivers may also be handled as program execution, but a few pieces of extra information must be entered explicitly in the request header.

Although the programming is contained within one admittedly long source file, the interface is presented as a library, partly to allow for future division, but also because its linkage may be automated by placing an

```
INCLUDELIB crtdrvr.lib
```

directive in the DRIVER.ASM file given earlier. Should this be done, the linker's /NOE (/NOEXTDICTIONARY) switch will be needed if another `DrvrProcInit` procedure is used instead of the one in the CRTDRVR library.

```
; CRTDRVR.ASM

;   This module prepares the interface between device driver initialization
;   and the C run time. The intention is to match the two sufficiently well
;   to allow driver initialization to be handled in C through the main ()
;   function, as if writing a normal application program.

.NOLIST
  INCLUDE          arena.inc
  INCLUDE          ascii.inc
  INCLUDE          driver.inc
  INCLUDE          psp.inc
  INCLUDE          sft.inc
  INCLUDE          sysvars.inc
.LIST

.MODEL                       SMALL, FARSTACK

;   The Keep C segmentation model for resident C programming is adopted,
;   but the segment definitions provided in the KEEPSEGS.INC file must be
;   modified to accommodate paragraph-alignment of a simulated PSP and
;   memory arena header.

KEEP_BTEXT                   SEGMENT PARA COMMON 'CODE'
KEEP_BTEXT                   ENDS

KEEP_TEXT                    SEGMENT PARA PUBLIC 'CODE'
KEEP_TEXT                    ENDS

KEEP                         SEGMENT WORD PUBLIC 'FAR_DATA'
KEEP                         ENDS

KEEP_GROUP                   GROUP     KEEP
```

```
;   Paragraph alignment is also needed for a simulated environment and
;   accompanying memory arena header, best handled by declaring a private
;   segment within DGROUP. The environment cannot be in the BSS area, as
;   explained below.

_ENV                         SEGMENT PARA PRIVATE 'DATA'
_ENV                         ENDS

DGROUP                       GROUP     _ENV

;   Some measure of the program's stack (whose size may be set by the
;   linker) is required. Create a paragraph-aligned fence to follow the
;   default STACK segment. Note that in the presence of DOSSEG ordering,
;   the fence must be in DGROUP (as must the default stack). This method
;   is ignorant of changes made to the stack by the EXEHDR utility.

.STACK                       0000h
.DOSSEG

ENDSTACK           SEGMENT PARA COMMON 'STACK'
ENDSTACK           ENDS

DGROUP                       GROUP     ENDSTACK

;   =========================================================================

.CODE                        KEEP_TEXT

;   The interface with the C run-time simulates the entry conditions of a
;   normal DOS executable. Most important is a PSP, set up so that control
;   can be regained when the high-level component of the program terminates
;   by the usual means.

;   The driver is allowed to overwrite its initialization code and data, but
;   the PSP must be valid at termination, so place the fake PSP in resident
;   memory (paragraph-aligned). The code segment KEEP_TEXT is selected to
;   avoid conflict with the scheme for repositioning the resident data.
;   This situation is not entirely satisfactory and an alternative ought to
;   be implemented in a future version.

FakePSPArena                 ARENA_HEADER       {ARENA_LAST, 0000h, 0000h,,
"CRTDRVR"}
FakePSP            PSP       <>

;   The following pieces of related data are needed before and after the
;   jump to the C run-time. They too must be placed in a resident segment.
;   Access is easier if they lie in the same segment as the PSP, rather than
;   in a KEEP_GROUP segment.

lpDTA                           dd          ?
lpInt22h            dd          ?
lpSysinitStack      dd          ?
spSysinitPSP                    dw          ?

;   Similar considerations apply to the installation of a System File Table
;   for use with File Control Blocks.

fcbsft_header                   SFT_HEADER          {0FFFFFFFFh, 0001h}
fcbsft                          SFT         <>
lpFCBSFTptr                     dd          00000000h

;   ========================================================================

_ENV                            SEGMENT

;   It is not essential that a program have an environment, but one is
;   simulated to enable the C run-time to provide a pathname as argv [0].

;   Note that the environment cannot be declared as uninitialized data,
;   because the C run-time would clear it before extracting the pathname.

;   Both the PSP and environment are accompanied by fake memory arena
;   headers, but the illusion is not carried further (just far enough, for
;   instance, to support routines that inspect the arena header to obtain
;   a practical limit on the environment size).

FakeEnvArena                    ARENA_HEADER        {ARENA_LAST, 0000h, 0000h,,
"CRTDRVR"}
FakeEnv             db          0100h DUP (?)

_ENV                            ENDS

;   ========================================================================

.DATA?

;   Segment addresses for the fake environment and PSP are needed
;   only before the transfer to the C run-time and may therefore be declared
;   as uninitialized data:
```

```
spFakeEnv        dw      ?
spFakePSP        dw      ?

;   ===========================================================================

.DATA

;   Segment address giving an upper bound on memory available to the device
;   driver must be initialized because access is needed before the
;   addressability of the uninitialized data area has been established:

spTop                       dw       0000h

;   ===========================================================================

.CONST

;   Fatal error messages:

FatalError                  db       CR, LF,
                                     "CRTDRVR Fatal error: ",
                                     "$"

BadVersion                  db       "DOS version 5 or higher required",
                                     CR, LF,
                                     "$"

OutOfMemory                 db       "Insufficient memory for driver",
                                     CR, LF,
                                     "$"

;   ===========================================================================

.FARDATA                    KEEP

;   A Boolean flag in a resident data segment to allow the high-level
;   component to determine whether it is being executed for device driver
;   initialization or as a normal program:

                            PUBLIC   C bLoadedAsDriver
bLoadedAsDriver  db         00h

;   ===========================================================================

.CODE                       KEEP_TEXT
```

```
DrvrProcInit                PROC      NEAR C PUBLIC lpReqHdr:DWORD

;   The DrvrProcInit function is the C-language entry point required by
;   the DRIVER.OBJ module for the initialization command. Although this
;   entry point is needed only once, it lies in and is called from the
;   resident code segment KEEP_TEXT, for the sake of uniformity.

;   Transfer the initialization request to non-resident code.

                            ASSUME    ds:KEEP_GROUP

                            push      word ptr [lpReqHdr + 02h]
                            push      word ptr [lpReqHdr]
                            call      far ptr NonResidentInit
                            pop       bx
                            pop       bx
                            ret

DrvrProcInit                ENDP

;   =====================================================================

.CODE

NonResidentInit    PROC     FAR C PRIVATE USES ds di si, lpReqHdr:DWORD

;   While ds still addresses the resident data group, set a flag to show that
;   the program has been loaded as a device driver.

                            mov       [bLoadedAsDriver],0FFh

;   The device driver will have been loaded into memory as an overlay,
;   without regard to the program's memory requirements in excess of its
;   disk image. The validity of addresses assigned to uninitialized data
;   must therefore be established before attempting access.

;   In a feature new to DOS 5, an upper limit to available memory is passed
;   as a far pointer in the request header—the same field in which the
;   driver is expected to return the address of the first byte of memory it
;   will not require. The provision of this field depends on the real DOS
;   version reported by DOS function 3306h, not the usual function 30h.

                            lds       si,[lpReqHdr]
```

```
                                mov     ax,DGROUP
                                mov     es,ax

                                ASSUME ds:NOTHING, es:DGROUP, si:PTR REQUEST_HEADER

                                mov     ax,3306h
                                xor     bx,bx
                                int     21h
                                mov     dx,OFFSET BadVersion
                                cmp     bl,05h
                                jb      error

                                mov     ax,word ptr [si].lpEndOfMemory
                                mov     cl,04h
                                shr     ax,cl
                                add     ax,word ptr [si + 02h].lpEndOfMemory
                                mov     es:[spTop],ax

                                cmp     ax,ENDSTACK
                                jnb     @f

                                mov     dx,OFFSET OutOfMemory
error:
                                push    ds
                                push    es
                                pop     ds
                                push    dx
                                mov     ah,09h
                                mov     dx,OFFSET FatalError
                                int     21h
                                pop     dx
                                mov     ah,09h
                                int     21h
                                pop     ds

                                jmp     far ptr fail
@@:

;   Prepare the simulated environment:

;       The device driver command line, whose address is passed in the
;       request header, starts with the first non-separator after the DEVICE
;       or DEVICEHIGH directive (including its optional SIZE parameter),
;       i.e., the first character of the device driver's pathname.
;         In a DEVICE statement, the end of the pathname is marked by any
;         character < 20h, space, comma, forward slash, semicolon, or equals
;         sign, but by the time SYSINIT is ready to initialize the driver,
;         this separator will have been converted to a space. DEVICEHIGH
;         directives are treated slightly differently in that the separator is
;         preserved, but note that the only control characters regarded as
;         separators are the null byte, tab and carriage-return. In any case,
;         all lower-case letters are capitalized and the line-feed is retained
;         as the terminator.

;         The C run time (as may any program) expects the pathname to have
;         been appended to the environment by the DOS "load and execute"
;         function. Recall that the environment consists of consecutive
;         ASCIIZ strings with an extra null to terminate the list. DOS
;         follows this with the word 0001h and then the pathname as an ASCIIZ
;         string.

;         No attempt is made to canonicalize the pathname—either by DOS or
;         the following code. Note, however, that more than a few
;         professionally-written programs—among them WIN.COM—assume the
;         presence of at least one backslash in the pathname they take from
;         the environment.

                                  lds       si,[si].lpCommandLine
                                  ASSUME    si:NOTHING

                                  mov       di,OFFSET FakeEnv

                                  xor       ax,ax
                                  stosw
                                  mov       al,01h
                                  stosw
@@:
                                  lodsb
  FOR                   separator, <NULL, TAB, LF, CR, " ", ",", "/", ";", "=">
                                  cmp       al,separator
                                  jz        @f
  ENDM
                                  stosb
                                  jmp       @b
@@:
                                  xor       al,al
                                  stosb

;   Save the pointer into the device driver command line—the remainder
;    must yet go into the PSP.

                        push      ds
                        push      si

;    The fake PSP and environment are both paragraph-aligned and have
;    equivalent segment addresses. These are needed for the arena headers
;    and again later, when the fake PSP is activated.

                        mov       cl,04h
                        mov       bx,OFFSET FakeEnv
                        shr       bx,cl
                        mov       dx,es
                        add       bx,dx
                        mov       es:[spFakeEnv],bx

                        mov       dx,KEEP_TEXT
                        mov       ds,dx
                        ASSUME    ds:KEEP_TEXT

                        mov       ax,OFFSET FakePSP
                        shr       ax,cl
                        add       ax,dx
                        mov       es:[spFakePSP],ax

;    Initialize the arena header for the environment.

                        mov       di,OFFSET FakeEnvArena.spOwner
                        stosw
                        mov       word ptr es:[di],SIZEOF FakeEnv SHR 4

;    Fill in the uninitialized fields in the arena header for the PSP.

                        mov       [FakePSPArena.spOwner],ax
                        mov       dx,ax

                        mov       si,es:[spTop]
                        sub       ax,si
                        neg       ax
                        mov       [FakePSPArena.wParagraphs],ax

;    Clone the current PSP (set up by the SYSINIT module of IO.SYS, which
;    loaded the device driver) and activate the copy. This is all done by
;    DOS function 55h—registers dx and si are expected to contain a segment
;    address at which to create the child process and another which delimits
;   the available memory.

                            mov     ah,55h
                            int     21h

;   Set the environment pointer in the fake PSP.

                            mov     [FakePSP.spEnvironment],bx

;   Copy the remaining command-line arguments to the PSP, taking care not
;   to exceed the available space. The command line in the PSP must be
;   terminated by a carriage-return and preceded by a byte count.

                            push    ds
                            pop     es
                            mov     di,OFFSET FakePSP.CommandLine

                            pop     si
                            pop     ds

                            ASSUME  ds:NOTHING, es:KEEP_TEXT
@@:
                            lodsb
                            cmp     al,LF
                            jz      @f
                            cmp     di,OFFSET FakePSP + SIZEOF PSP - 1
                            jnb     @f
                            stosb
                            jmp     @b
@@:
                            push    es
                            pop     ds
                            ASSUME  ds:KEEP_TEXT

                            mov     byte ptr [di],CR
                            mov     ax,OFFSET FakePSP.CommandLine
                            xchg    ax,di
                            sub     ax,di
                            dec     di
                            mov     [di],al

;   No set of SFTs dedicated for use with FCBs will exist yet, but as part
;   of the normal clean-up operations on process termination, DOS will try
;   to close outstanding FCBs—without checking that a set of FCB-SFTs
;   actually exists (after all, DOS has no intention of using them before
;   it is ready to run applications, by which time a suitable set will have
;   been constructed).

                            mov     ah,52h
                            int     21h
                            add     bx,SYSVARS.lpFCBSFTs
                            mov     word ptr [lpFCBSFTptr],bx
                            mov     word ptr [lpFCBSFTptr + 02h],es

                            mov     ax,es:[bx]
                            mov     dx,es:[bx + 02h]
                            or      ax,dx
                            jnz     @f

                            mov     es:[bx],OFFSET fcbsft_header
                            mov     es:[bx + 02h],ds
@@:

;   Set the DTA to the command line area of the PSP, as would normally be
;   done by DOS function 4B00h. Note, as more than a curiosity, that the
;   DTA address is not preserved by DOS function 4B00h.

                            mov     ah,2Fh
                            int     21h
                            mov     word ptr [lpDTA],bx
                            mov     word ptr [lpDTA + 02h],es

                            mov     ah,1Ah
                            mov     dx,di
                            int     21h

;   When the child process terminates, the parent receives control at an
;   address taken from a field at offset 0Ah in the child's PSP. DOS
;   function 55h (used to create the child PSP) will have set this
;   termination address to the current value of int 22h in the interrupt
;   vector table. Ordinarily, DOS function 55h is called internally by
;   the "load and execute" function 4Bh, which then points both the PSP
;   termination field and the int 22h vector to the instruction immediately
;   after the "load and execute" call. Simulate this by redirecting the
;   termination pointer and the int 22h vector to an instruction inside this
;   module (but in the resident code segment).

                            xor     ax,ax
                            mov     es,ax
                            ASSUME  es:NOTHING
```

```
                mov     di,22h * 4

                mov     ax,es:[di]
                mov     word ptr [lpInt22h],ax
                mov     ax,es:[di + 02h]
                mov     word ptr [lpInt22h + 02h],ax

                mov     ax,OFFSET CRTreturn
                mov     dx,SEG CRTreturn

                mov     es:[di],ax
                mov     es:[di + 02h],dx

                mov     word ptr [FakePSP.lpInt22h],ax
                mov     word ptr [FakePSP.lpInt22h + 02h],dx

;   The stack is restored from an address which would have been saved in the
;   parent's own PSP as a normal consequence of the parent's issuing an
;   int 21h instruction (typically the one to load the child program). The
;   address is saved by the DOS kernel only after the caller's CPU registers
;   have been pushed onto the stack for retrieval at termination.

                pushf
                push    dx
                push    ax

                push    es
                push    ds
                push    bp
                push    di
                push    si
                push    dx
                push    cx
                push    bx
                push    ax

                mov     ax,[FakePSP.spParentPSP]
                mov     [spSysinitPSP],ax
                mov     es,ax

                mov     ax,sp
                xchg    ax,word ptr es:[PSP.lpStack]
                mov     word ptr [lpSysinitStack],ax
                mov     ax,ss
                xchg    ax,word ptr es:[PSP.lpStack + 02h]
```

```
                        mov     word ptr [lpSysinitStack + 02h],ax

;   Reproduce the normal .EXE entry conditions expected by the _astart entry
;   point to the C run-time:

;       Address the simulated PSP with both ds and es.

                        mov     ax,DGROUP
                        mov     ds,ax
                        ASSUME  ds:DGROUP
                        mov     ax,[spFakePSP]
                        mov     ds,ax
                        mov     es,ax
                        ASSUME  ds:NOTHING, es:NOTHING

;       Point ss:sp to the top of the STACK segment.

                        mov     ax,ENDSTACK
                        mov     dx,STACK
                        sub     ax,dx
                        mov     cl,04h
                        shl     ax,cl

                        cli
                        mov     ss,dx
                        mov     sp,ax
                        sti

;       Register ax should contain a value describing the validity of the
;       drive elements in the two FCBs contained in the PSP. The fields
;       themselves are seldom used and the value in ax even less often.

                        mov     ax,0FFFFh

;   Transfer to _astart, but if the CRTKEEPC library is linked, execute its
;   initialization code first to rearrange segments as needed. Control will
;   be regained at the label CRTreturn in the following KEEP_TEXT segment.

                        EXTERN  C _astart:FAR
                        EXTERN  C InitKeepC (_astart):FAR

                        jmp     InitKeepC

NonResidentInit     ENDP
```

```
;   ======================================================================

.CODE                       KEEP_TEXT

CRTreturn       PROC        FAR C PRIVATE USES ds di si, lpReqHdr:DWORD

;   The prologue code for this procedure will have been set up by the
;   procedure NonResidentInit in the _TEXT segment—only the corresponding
;   epilogue sequence is required.

                            OPTION    PROLOGUE:NONE

;   Restore the int 22h vector and the stack address field in SYSINIT's PSP
;   to the values they had when this device driver was loaded. Note that
;   registers have been retrieved from the stack created earlier, so ds
;   still addresses KEEP_TEXT and es:di still points to the Interrupt
;   Vector Table location for int 22h.

                            ASSUME    ds:KEEP_TEXT, es:NOTHING

                            mov       si,OFFSET lpInt22h
                            movsw
                            movsw

                            mov       es,[spSysinitPSP]
                            mov       di,PSP.lpStack
                            mov       si,OFFSET lpSysinitStack
                            movsw
                            movsw
;   Restore the DTA.

                            push      ds
                            pop       es
                            lds       dx,[lpDTA]
                            ASSUME    ds:NOTHING, es:KEEP_TEXT

                            mov       ah,1Fh
                            int       21h

;   If an FCB-SFT was installed for the child, remove it.

                            lds       bx,es:[lpFCBSFTptr]

                            cmp       word ptr [bx],OFFSET fcbsft_header
                            jnz       @f
```

```
                        mov     ax,es
                        cmp     word ptr [bx + 02h],ax
                        jnz     @f

                        xor     ax,ax
                        mov     [bx],ax
                        mov     [bx + 02h],ax
@@:

;   Prepare to set values in the request header. Use the same registers as
;   in the NonResidentInit procedure.

                        lds     si,[lpReqHdr]
                        ASSUME  si:PTR REQUEST_HEADER

;   Obtain the child process's exit codes.

                        mov     ah,4Dh
                        int     21h

;   If the program stayed resident, retrieve the break address from the
;   PSP - DOS function 31h adjusts the "top of memory" field in the PSP
;   if it is in fact able to resize the PSP's memory block.

                        cmp     ah,03h
                        jnz     @f

                        mov     ax,es:[FakePSP.spCeiling]
                        mov     word ptr [si].lpEndOfMemory,0000h
                        mov     word ptr [si + 02h].lpEndOfMemory,ax

                        mov     ax,D_DONE

                        jmp     done
@@:

;   For the non-resident terminations, set the flag for the system error
;   message (introduced in DOS 4) if the cause of termination was a DOS
;   exception (Ctrl-C or a critical error) or if the child's return code
;   is >= 80h.

                        or      ah,ah
                        jnz     fail
                        or      al,al
                        jns     @f
```

```
fail::
                        mov      [si].wMessageFlag,0001h
@@:

;   Abandon the device-driver installation by indicating that none of the
;   program, from the very first byte onward, is required.

                        mov      word ptr [si].lpEndOfMemory,0000h
                        mov word ptr [si+02h].lpEndOfMemory,KEEP_BTEXT

;   Block devices are also rejected if they return a count of no units to
;   support. Note that for neither type of device driver does DOS consult
;   the status field for an error flag, although the published reference
;   states that the error bit should be set to indicate failure.

                        mov      [si].cUnitsSupported,00h

                        mov      ax,D_ERROR OR D_DONE OR D_GENERAL_FAILURE
done:
                        ret

CRTreturn       ENDP

END
```

An Example—Loading a Resident Program as a Device Driver

The small example program INT15MEM which closed the chapter on Resident Programming in C may now be relinked as a device driver so that the effect of trapping int 15h to obtain extended memory may be examined in the presence of HIMEM.SYS. Simply change the LINK command in the INT15MEM makefile from

```
$(LINK) crtkeepc.lib+handler+main,int15mem.exe;
```

to

```
$(LINK) driver+crtdrvr.lib+crtkeepc.lib+handler+main,int15mem.exe;
```

and remake the executable. The first three modules have general applicability: DRIVER.OBJ provides the device-driver front-end; CRTDRVR.LIB interfaces device-driver initialization with the program's normal entry point; CRTKEEPC.LIB supplies a segmentation model for resident programming. The last two modules are the substance of the INT15MEM program: HANDLER.OBJ contains the resident program material and MAIN.OBJ initializes the TSR. Both modules specific to INT15MEM have been written in C with no special effort out of all proportion to the fact that INT15MEM is only an investigative gadget. *With no extra coding*, the program—written for normal operating conditions—can also be run as the initialization phase of a DOS device driver.

INT15MEM may now be loaded before and after the installation of HIMEM. As noted earlier, HIMEM itself obtains extended memory by trapping int 15h to convince subsequent programs that less memory remains. The hope is that programs will prefer to access extended memory through the XMS interface supplied by HIMEM, because of its greater flexibility and independence from the order of allocation, but HIMEM must also co-operate with other (probably older) programs that use int 15h directly. This interplay is discussed in Part III of this book, devoted to extended memory and the XMS. Readers who want to investigate a little before passing to Chapter 10 (where INT15MEM serves to demonstrate a potentially serious conflict between int 15h allocation and the XMS as implemented by HIMEM.SYS) may care to begin by musing over why INT15MEM reports (as expected) that no extended memory is available via int 15h when loaded through the CONFIG.SYS file

```
device=himem.sys
device=int15mem.exe
dos=high
```

but thinks differently if **dos=low** is used instead.

An Example—A Real Device Driver

The implementation of the INT15MEM program demonstrates the adaptation of a simple int 15h function to serve as an allocation method, but does absolutely nothing with the block of extended memory that it sets aside. Some difficulty attends the use of extended memory by programs that operate the processor in real mode, and it is instructive to consider this while also taking the opportunity to show how the tools developed here may be used to write a device driver that interacts with the DOS kernel through more than just the initialization function.

Extended Memory Access

Real mode does not permit formal addressing of memory much beyond the first megabyte, and although the realities of processor design allow some licence to be taken with the formal concept, assume for present purposes that data transfers to and from extended memory must be conducted in protected mode. On this basis the BIOS provides int 15h function 87h (Move Extended Memory Block) to step up to protected mode, move the data on the caller's behalf, and then return to real mode.

The protected-mode environment constructed for the move is necessarily minimal. Service routines for hardware interrupts might not be suitable for operation outside of real mode and interrupts are in fact disabled while the BIOS function works in protected mode. This is a serious disadvantage; with a CPU clocked at 16MHz, merely reading a track of 8 sectors from a RAM disk in extended memory takes long enough to lose data from a serial port at 9600 baud. Matters are even worse for the 80286 processor, where the return to real mode adds a substantial delay while the CPU is halted and restarted. Alternative schemes are a practical necessity and were sought from the outset, first to avoid the 80286 penalty and then to allow interrupts during the move, but they are fairly elaborate and to some extent hardware-dependent. Any implementation ought to be provided as communal property, which is another compelling motivation for the XMS interface examined over the several chapters of Part 3.

The int 15h facility sets up an Interrupt Descriptor Table (IDT) with gates to routines capable of fielding any processor exceptions that might occur during the move and a Global Descriptor Table (GDT) that supports a null selector and segments for the BIOS code, data and stack (usually the caller's), as well as segments for the source and destination. The same IDT will serve for all instances and may therefore be held in ROM, but the GDT is supplied by the caller, with two descriptors already filled in for the source and destination. The function is passed AH = 87h, CX equal to the number of words to transfer, and ES:SI addressing 48 bytes to use as the GDT. Descriptors for the source and destination are GDT entries 2 and 3, respectively, corresponding to ring 0 selectors 0010h and 0018h. They should contain the correct linear addresses and some suitable limit, along with access rights appropriate for a read/write data segment. If successful, the function should return with a clear carry flag and ah = 00h. Failure may occur because of a memory parity error, interruption by a processor exception (for example, if either segment limit is overrun), or because the A20 line could not be enabled. In all these cases, the carry flag will be set and ah will contain a non-zero error code.

In a sense the extended memory to which this BIOS function provides access is a peripheral device of which DOS is not normally aware but which could be made available to the file system through a device driver. A block device driver would effectively simulate a disk, but would also entail more work than an example warrants and, in any case, a perfectly good RAMDRIVE exists already.

File Emulation

Consider instead a character device driver. It may be opened under its logical device name, after which data may be read from it or written to it, using normal DOS functions throughout. In much the same way that the screen cursor is moved ahead as successive characters are displayed, characters written to the new device will be copied to successive locations within the block of extended memory. The read/write position within the block is control data for the device and IOCTL support may be provided for manipulating this "file pointer" if sequential access seems too limiting.

Readers are encouraged to persist with this possibly unfamiliar notion, because one of the modules presented here will be incorporated unchanged into a disk duplication utility developed in Part 4, along with another module, created here but rewritten in Part 3 to use the more flexible XMS interface. The key to writing code capable of easy adaptation is to draw the logical boundaries between modules, so let's begin with a source file which recognizes that the essential nature of the file emulation is independent of the methods used for extended memory allocation and access.

```
/* SEQFILE.C */

/*  This file contains routines for emulating sequential file I/O with a
    character device driver using extended memory for storage. */

#include    <driver.h>
#include    <standard.h>

/*  The "file" is described only by a pointer to the position for the next
    read or write operation and by an end-of-file marker. */

DWORD dwFilePtr = 0, dwFileEnd = 0;

/*  Services dependent on the nature of extended memory access are obtained
    externally through the following functions. */

extern DWORD AdjustRequest (BYTE, DWORD);
extern WORD Close (void);
extern BOOL MoveBlock (BYTE, BYTE _far *, DWORD, WORD);
extern WORD Open (void);

/*  ==================================================================  */

/*  There is much similarity between reading and writing, so the device
    driver I/O functions call the following routines, passing a command
    number as the first argument to support the little differentiation
    required. Both functions return a device driver status word. */

static WORD ReadWrite (BYTE, BYTE _far *, WORD _far *);
static WORD ReadWriteSingle (BYTE, BYTE _far *);

/*  The first function—the more general of the two—is able to move an
    arbitrary amount of data. In addition to the device-driver status, it
    must also indicate the number of bytes actually transferred, which is
    why the corresponding argument is an address. */

static WORD ReadWrite (BYTE command, BYTE _far *lpBuffer, WORD _far *lpCount)
{
    register WORD numcopied = 0, status = 0;
    WORD numwanted;
```

```
/*   It might not be possible to accommodate the whole request. This is
     determined by an external function, since the details may depend on
     the method used to obtain the extended memory block.

     A choice exists over the treatment of requests that can be filled
     only partially. DOS does not copy data to a disk file if space is
     not available to complete the operation, but it might be preferred
     here to truncate requests, leaving it to the caller to decide on
     further action—this is in fact the behavior adopted here, though
     it does mean that in some cases, an incomplete transfer may be
     reported as successful at user level. */

numwanted = (WORD) AdjustRequest (command, *lpCount);

if (numwanted == 0) status = D_DONE;
else {

     /*  Both the int 15h service and the more general XMS move data in
         words, so special provision must be made for odd bytes. In the
         meantime, move as much data as possible with one call. If
         successful, advance the various pointers and counters, but do
         not set the DONE bit if a spare byte remains to be transferred.

         The data transfer is conducted externally.  Note that the
         natural objects for describing the move are the far pointer to
         the caller's buffer in memory addressable from real mode and an
         offset into the simulated file. */

     if (numwanted > 1) {
         if (MoveBlock (command, lpBuffer, dwFilePtr, numwanted & ~1)) {
               lpBuffer += numwanted & ~1;
               dwFilePtr += numwanted & ~1;
               numcopied += numwanted & ~1;
               if (numcopied == numwanted) status = D_DONE;
         }
         else status = D_ERROR | D_DONE | D_GENERAL_FAILURE;
     }

    /*  If the request has not yet been completed, then either a single
        byte must be transferred or a spare byte remains from a larger
        operation. Either way, the operation is conducted by a separate
        function to lend support for the Non-destructive Read service. */

     if (NOT (status & D_DONE)) {
```

```
                status = ReadWriteSingle (command, lpBuffer);
                if (NOT (status & D_ERROR)) {
                     dwFilePtr ++;
                     numcopied ++;
                }
           }
      }

      /*    If data has been written past the end-of-file marker, then update
            the marker. */

      if (dwFilePtr > dwFileEnd) dwFileEnd = dwFilePtr;

      /*  Return the caller's information - the number of bytes successfully
          copied is returned indirectly. */

      *lpCount = numcopied;
      return (status);
}

/*  The following function moves a single byte and has the same prototype
    as the ReadWrite function except that the argument related to the size
    of the transfer is not required. Unlike the preceding function, this one
    does not adjust the "file pointer" after a successful transfer. It may
    be assumed that one byte has indeed been copied if the function returns
    with the ERROR bit clear. */

static WORD ReadWriteSingle (BYTE command, BYTE _far *lpBuffer)
{
      struct {
            BYTE lowbyte;
            BYTE highbyte;
      } temp;
      DWORD xmtemp;
      register WORD first;

      /*    To transfer a single byte, two bytes must be read into a holding
            buffer so that the one wanted may be either fished out or updated.
            A small complication is introduced by boundary effects: The first
            byte in the block can only be paired with the one immediately above
            it, the last byte only with the one below it.

            Testing is easier if the first byte is regarded as exceptional, so
            that in all other cases, the byte of interest is paired with the
            one immediately before it in the extended memory block. */
    xmtemp = dwFilePtr;
    if (dwFilePtr == 0) first = TRUE;
    else {
          first = FALSE;
          xmtemp --;
    }

    /*    Read a pair of bytes from extended memory into the temporary area. */

    if (MoveBlock (D_READ, (BYTE _far *) &temp, xmtemp, 2) == 0)
          return (D_ERROR | D_DONE | D_GENERAL_FAILURE);

    /*    For a device-driver read operation, simply copy the desired byte to
          the caller's buffer. For a write operation, update the byte in the
          temporary buffer, then move the pair back to extended memory. */

    if (command == D_READ) *lpBuffer = first ? temp.lowbyte : temp.highbyte;
    else {
          *(first ? &temp.lowbyte : &temp.highbyte) = *lpBuffer;
          if (MoveBlock (D_WRITE, (BYTE _far *) &temp, xmtemp, 2) == 0)
              return (D_ERROR | D_DONE | D_GENERAL_FAILURE);
    }
    return (D_DONE);
}

/*  ==================================================================  */

/*  Device driver functions - in the form required for DRIVER.ASM  */

/*  Read and write functions transfer the data between the caller's buffer
    and a region of extended memory beginning some distance into the
    extended memory block. They have much in common, so in both cases,
    extract relevant fields from the request packet and pass them to a
    common function. */

WORD DrvrProcRead (struct REQUEST_HEADER _far *lpReqHdr)
{
    return (ReadWrite (D_READ, lpReqHdr -> lpBuffer, &(lpReqHdr -> wCount)));
}

WORD DrvrProcWrite (struct REQUEST_HEADER _far *lpReqHdr)
{
    return (ReadWrite (D_WRITE, lpReqHdr -> lpBuffer, &(lpReqHdr -> wCount)));
}
```

```
/*  The Write With Verify command is treated no differently from the normal
    Write command. */

WORD DrvrProcWriteWithVerify (struct REQUEST_HEADER _far *lpReqHdr)
{
    return (DrvrProcWrite (lpReqHdr));
}

/*  For the Non-destructive Read, first check that a byte is available, then
    call the separate function used for picking up spares. The Input Status
    command is the same but does not return the waiting character. Note,
    however, that DOS never issues this command: int 21h function 4406h --
    described as Input Status -- is actually a Non-destructive Read at device-
    driver level. */

WORD DrvrProcNondestructiveRead (struct REQUEST_HEADER _far *lpReqHdr)
{
    if (AdjustRequest (D_READ, 1) == 1)
           return (ReadWriteSingle (D_READ, &(lpReqHdr -> cCharWaiting)));
    else return (D_ERROR | D_DONE | D_GENERAL_FAILURE);
}

WORD DrvrProcInputStatus (struct REQUEST_HEADER _far *lpReqHdr)
{
    if (AdjustRequest (D_READ, 1) == 1) return (D_DONE);
    else return (D_ERROR | D_DONE | D_GENERAL_FAILURE);
}

WORD DrvrProcOutputStatus (struct REQUEST_HEADER _far *lpReqHdr)
{
    if (AdjustRequest (D_WRITE, 1) == 1) return (D_DONE);
    else return (D_ERROR | D_DONE | D_GENERAL_FAILURE);
}

/*  ------------------------------------------------------------------------  */

/*  The Open Device command may be used to ensure that the file pointer is
    reset to zero whenever the device is opened. Among other things, this
    supports storage and retrieval across successive openings.

    The followng function for closing the device truncates the "file" at
    the position of the last read or write.

    In both cases, the function is also passed externally to support
    operations that depend on the nature of the extended memory allocation.

    No attempt is made here to address the question of multiple ownership. */

WORD DrvrProcOpenDevice (struct REQUEST_HEADER _far *lpReqHdr)
{
    dwFilePtr = 0;
    return (Open ());
}

WORD DrvrProcCloseDevice (struct REQUEST_HEADER _far *lpReqHdr)
{
    dwFileEnd = dwFilePtr;
    return (Close ());
}

/*  ----------------------------------------------------------------  */

/*  The IOCTL interface—intended for passing control data to and from the
    device—may be used to get and set the "file pointer". In the simple
    implementation which follows, this pointer—a 32-bit offset into the
    extended memory block—is passed as a DWORD in the IOCTL buffer. */

WORD DrvrProcIOCTLRead (struct REQUEST_HEADER _far *lpReqHdr)
{
    /*    If the transfer count is the right size, copy the "file pointer"
          into the buffer. */

    if (lpReqHdr -> wCount == sizeof (dwFilePtr))
          *((DWORD _far *) lpReqHdr -> lpBuffer) = dwFilePtr;
    else lpReqHdr -> wCount = 0;
    return (D_DONE);
}

WORD DrvrProcIOCTLWrite (struct REQUEST_HEADER _far *lpReqHdr)
{
    DWORD temp;

    /*    If the caller indicates that the buffer contains exactly four bytes
          of control data, interpret these bytes as a new value for the file
          pointer. */

    if (lpReqHdr -> wCount == sizeof (dwFilePtr)) {

          /*  Accept a lower value for the file pointer but allow an advance
```

```
            only if it would be possible to write that far into the file. */

        temp = *((DWORD _far *) lpReqHdr -> lpBuffer);
        if (temp <= dwFilePtr) dwFilePtr = temp;
        else {
            temp -= dwFilePtr;
            if (AdjustRequest (D_WRITE, temp) == temp) dwFilePtr += temp;
            else lpReqHdr -> wCount = 0;
        }
    }
    else lpReqHdr -> wCount = 0;
    return (D_DONE);
}
```

Initializing the Device

INT15MEM's initialization in its MAIN.C module must be amended slightly to record (in resident memory) the base and size of the extended memory block allocated by trapping int 15h function 88h (Get Extended Memory Size), but this is a simple matter of adding

```
extern DWORD _RESIDENT dwExtMemBase, _RESIDENT dwExtMemSize;
```

at the top of the file and

```
/*  Calculate linear addresses, etc, for later. */

dwExtMemBase = (DWORD) (1024 + MemLeftInKB) << 10;
dwExtMemSize = (DWORD) MemToAllocateInKB << 10;
```

in an appropriate place toward the end of the `main` function.

Managing the Extended Memory Block

This block of extended memory is no longer inert. In response to a read or write request from the DOS kernel, the INT15MEM device driver must be able to transfer data to or from extended memory. The SEQFILE module handles most of the control problem, but there remain a few tasks tied closely to the int 15h interface. (There is a different version of TRANSFER.C that uses XMS, presented with the XMSFILE program later in this book.)

```
/* TRANSFER.C */

/*  This file contains material specific to the int 15h method of moving
```

```
    data to and from extended memory. It accompanies the character device
    driver emulation of a sequential file in extended memory. */

#include     <driver.h>
#include     <standard.h>

/*  ==========================================================================  */

/*  The block's linear address and size must be recorded during device-
driver initialization (the responsibility of another module). */

DWORD dwExtMemBase = 0, dwExtMemSize = 0;

/*  Variables describing the file's I/O state are held externally. */

extern DWORD dwFilePtr, dwFileEnd;                  // in seqfile.c

/*  The BIOS provides an int 15h service for moving data to or from extended
    memory. It requires a Global Descriptor Table (GDT) with space for six
    segment descriptors, two of which are manipulated by the caller to
    specify source and destination. */

#pragma pack (1)

static struct SEGMENT_DESCRIPTOR {
    WORD limit;
    WORD base;
    BYTE base_middle_8;
    BYTE access_rights;
    BYTE extra_byte;
    BYTE base_high_8;
} gdt [6] = {0};

/*  ==========================================================================  */

/*  The function ToLinear converts a far pointer from segment:offset form to
    an equivalent linear address needed for a segment descriptor. */

static DWORD ToLinear (void _far *lp)
{
    /*   The calculation is essentially

             ((DWORD) FP_SEG (lp) << 4) + FP_OFF (lp)

         but must be modified, since the compiler calls a non-resident
```

```
          library function to perform bit shifts in 32-bit objects. This is
          not permissable in a resident module, but note that even without
          this restriction, the following algorithm generates more efficient
          code. */

    #define FP_SEG(lp)        ((WORD) (_segment) (lp))
    #define FP_OFF(lp)        ((WORD) (lp))

    return (MK_DWORD (FP_SEG (lp) >> 12, (FP_SEG (lp) << 4) + FP_OFF (lp)));
}

/*  The details of the int 15h facility for transferring data to and from
    extended memory are handled in the following function. This fills in
    the two descriptors allowed for the source and destination (using linear
    addresses passed as arguments to the function), then issues the int 15h
    instruction to move the data. No other function need have access to the
    descriptor table. */

BOOL MoveBlock (BYTE command, BYTE _far *lpBuffer, DWORD xmptr, WORD count)
{
    DWORD source, dest;

    if (command == D_READ) {
          source = dwExtMemBase + xmptr;
          dest = ToLinear (lpBuffer);
    }
    else {
          source = ToLinear (lpBuffer);
          dest = dwExtMemBase + xmptr;
    }

    gdt [2].limit = 0xFFFF;
    gdt [2].base = LOWWORD (source);
    gdt [2].base_middle_8 = LOWBYTE (HIGHWORD (source));
    gdt [2].access_rights = 0x93;
    gdt [2].extra_byte = 0;
    gdt [2].base_high_8 = HIGHBYTE (HIGHWORD (source));

    gdt [3].limit = 0xFFFF;
    gdt [3].base = LOWWORD (dest);
    gdt [3].base_middle_8 = LOWBYTE (HIGHWORD (dest));
    gdt [3].access_rights = 0x93;
    gdt [3].extra_byte = 0;
    gdt [3].base_high_8 = HIGHBYTE (HIGHWORD (dest));
    _asm {
            push    ds
            pop     es
            mov     si,OFFSET gdt    ; compiler preserves si
            mov     cx,count
            shr     cx,1
            mov     ah,87h
            clc
            int     15h
            jc       bad
            mov     ax,0001h         ; TRUE
            jmp     done
        bad:
            xor     ax,ax            ; FALSE
        done:
    }
}

/*  The function AdjustRequest is called to determine how much of a read or
    write request can actually be met. Its return value is necessarily less
    than or equal to the DWORD argument. */

DWORD AdjustRequest (BYTE command, DWORD numwanted)
{
    DWORD avail;

    /*    The block of extended memory obtained by trapping int 15h cannot be
          resized easily, so little adjustment is possible. Simply truncate
          requests to fit the available space. */

    avail = (command == D_READ ? dwFileEnd : dwExtMemSize) - dwFilePtr;
    if (numwanted > avail) return (avail);
    return (numwanted);
}

/*  ========================================================================  */

/*  No special steps need be taken on opening or closing the device. */

WORD Open (void)
{
    return (D_DONE);
}

WORD Close (void)
```

```
{
    return (D_DONE);
}
```

The Completed Driver

The driver's support for IOCTL and open/close functions is only recognized by the DOS kernel if the corresponding bits are set in the device driver attribute word. Given the ready availability of source code to the front-end module DRIVER.ASM, it is necessary simply to assemble a copy of that file with a command-line definition for the `_ATTRIBUTE_` symbol, providing either a numerical equate or a value constructed from symbols given in the DRIVER.INC file. The driver is assigned the logical device name INT15XXX as an equate for the `_NAME_` symbol. Thus, a complete makefile for the device driver version of INT15MEM.EXE is

```
!INCLUDE <standard.mak>
!INCLUDE <keepc.mak>

TARGETS : int15mem.exe

driver.obj : driver.asm
  $(ASSEMBLE) /D "_ATTRIBUTE_=IS_CHAR_DEVICE OR IOCTL_OK OR OCRM_OK" \
  /D _NAME_=INT15XXX driver.asm

handler.obj : handler.c
  $(COMPILE) $(KEEP) handler.c

main.obj : main.c

seqfile.obj : seqfile.c
  $(COMPILE) $(KEEP) seqfile.c

transfer.obj : transfer.c
  $(COMPILE) $(KEEP) transfer.c

#   Link with DRIVER.OBJ, CRTDRVR.LIB and CRTKEEPC.LIB

int15mem.exe : driver.obj handler.obj main.obj seqfile.obj transfer.obj
  $(LINK) driver+crtdrvr.lib+crtkeepc.lib+handler+main+seqfile+transfer, \
          int15mem.exe;
```

Now, after rebooting with a line similar to

```
device=int15mem.exe 64
```

in CONFIG.SYS, the 64K of extended memory that would previously have just been "eaten" can be used for temporary storage. The logical device simulates a file to the extent that the command

```
type transfer.c >int15xxx
```

saves the device-driver source file in extended memory and the reverse command

```
type int15xxx >tempfile
```

retrieves it into another file. The INT15XXX device acts as a sort of clipboard.

Unfortunately the COPY command is harder to support because by the time it opens its source and destination for the data transfer, it will have already opened and closed them as part of a technique for discovering their properties. Thus,

```
copy transfer.c int15xxx
```

behaves as expected, but the reverse command inadvertently truncates the "file" before attempting to retrieve its contents. It is not really possible to have a character device emulate a file convincingly. In particular, truncation by writing zero bytes cannot be supported because empty writes are not passed to character device drivers. Likewise, monitoring the real file pointer in the SFT is pointless, since DOS does not maintain it reliably for character device drivers (the file pointer is not advanced when devices are accessed through the so-called console functions).

Operating Modes for Devices

Character devices are operated in text mode unless deliberate action is taken to place them in binary mode. In text mode (as in the COPY and TYPE examples just given), the driver is asked to transfer data one byte at a time and the DOS kernel inspects each byte so that the read/write operation may be terminated on finding an end-of-file character. When a device is in binary mode, the DOS kernel lets the driver fill the caller's needs in one go. The distinction may be seen clearly here because the move to extended memory using int 15h function 87h attracts significant overhead. Note especially that because the BIOS function transfers data in word multiples, writing a single byte to the INT15XXX device requires two BIOS calls, doubling the penalty. If possible, all the data should be moved to extended memory with one call by operating the device in binary mode, as in the command

```
copy transfer.c /b int15xxx
```

which is very much faster than its text mode equivalent.

More flexibility is available to a program that wants to use the INT15XXX device for its own storage purposes and may therefore control the device at DOS function level instead of relying on the vaguaries of DOS's command-level support for character devices. The opened device may be placed in binary mode using int 21h function 4401h (Set Device Data), having first called int 21h function 4400h (Get Device Data) to establish that the handle does in fact refer to a character device rather than a file. Data may be read from and written to the device through DOS functions 3Fh and 40h as usual. Should it be necessary to manipulate the "file pointer," int 21h functions 4402h and 4403h (IOCTL Read and Write) are available to get and set it, respectively. These IOCTL functions correspond to those for normal I/O. Thus,

```
WORD _dos_ioctlwrite (int handle, void _far *buffer, WORD count, WORD *numwrt)
{
    WORD temp;

    _asm {
              push    ds
              mov     ax,4403h        //   instead of mov ah,40h
              mov     bx,handle
              mov     cx,count
              lds     dx,buffer
              int     21h
              pop     ds
              jc       done
              mov     temp,ax         //   temporary storage of transfer count,
    }                                 //   though inefficient, ensures
    *numwrt = temp;                   //   independence from memory model
    _asm {
              xor     ax,ax
         done:
    }
}
```

matches the `_dos_write` function in the Microsoft C small-model library. The meaning of the control data passed through this interface is a private matter between the driver and its clients. For simplicity, TRANSFER.C implements a scheme in which the 32-bit file pointer is passed as four bytes of data in the buffer. Attempting to set an invalid value for the pointer will be indicated by the driver's refusal to accept the control data; thus,

```
BOOL fake_seek (int handle, long fp)
{
    unsigned numwrt;

    if (_dos_ioctlwrite (handle, &fp, sizeof (fp), &numwrt) == 0
        AND numwrt == sizeof (fp)) return (TRUE);
    else return (FALSE);
}
```

returns TRUE if and only if the file pointer has been updated successfully.

The simulated file pointer is not quite the same as one maintained by DOS for an ordinary file, most obviously because it is subject to a different int 21h function. In many situations requiring temporary storage, it suffices that the pointer advance automatically as data is read or written and that it be rewound to the start when the file is next opened. In such circumstances, it may be a real benefit that data can be sent either to disk or extended memory using the same code. In any case, it is certainly helpful to be able to play with these possibilities in C.

PART 3

Extended Memory Management

The Extended Memory Specification (XMS) arbitrates the use of memory lying outside the conventional DOS domain of the first 640K. Of particular concern as computing hardware increases in sophistication is the potentially vast amount of extended memory that may be installed above the 1MB boundary of the 8086 address space. The 80286 and more modern processors can form addresses with more than the 20 bits to which the 8086 is limited, but not by following the real-mode rule for combining 16-bit segments and offsets. With only a few exceptions, the protected mode of these newer processors is required to get at extended memory, putting this practically unbounded region outside the range of normal DOS memory management.

The humble 8086 and 8088 are still in use, and DOS exists in a basic form that does not encumber users of these machines with provisions for computing power they do not possess. Furthermore, a real-mode environment is ideal for many purposes requiring moderate resources and only minimal assistance from an operating system. The XMS acts as a supplement, presenting the additional resource of extended memory without the trappings that may make a protected-mode operating system seem extravagant. When extra power is needed, the XMS is an essential agent in the smooth transition to protected mode, offering the means through which the protected-mode software may preserve extended memory already being used by real-mode programs. In short, the XMS is the means by which DOS may usefully serve as a launching pad to protected mode but not lose touch with its basic configuration in real mode.

The Interface in Outline

An XMS implementation sufficient to meet the needs of programs written for real mode on machines with more than 1MB of addressable memory is provided by the HIMEM.SYS device driver, which is distributed as standard with DOS 5 and higher but usable with version 3.00 or higher. Strictly speaking, the program is not a device driver. It supports no device driver functions other than initialization, but is coded in that form because uniform management of extended memory must be available to all programs, no matter how early they are loaded.

The interface through which programs may obtain XMS services is carried by the increasingly overworked multiplex facility on int 2Fh, but the individual services are not implemented directly as int 2Fh functions. Instead, the combination of int 2Fh functions shown in Table 1 may establish the existence in memory of an XMS server such as HIMEM and return a calling address for the client to use in all further communication.

Table 1: Documented Int 2Fh Support for the XMS

Int 2Fh function 4300h—XMS Availability Check

INPUT:

AX 4300h

SUCCESS:

al 80h

Int 2Fh function 4310h—Get XMS Calling Address

INPUT:

AX 4310h

OUTPUT:

es:bx address of XMS calling point

A C-language function that issues the necessary int 2Fh instructions to obtain the XMS calling address will be used by almost all programs presented in this part of the book.

```
/* GETXMS.C */

/*  The following C function returns the address of the XMS calling point,
    if an XMS server is installed. Otherwise, it returns a null pointer. */

void (_far *GetXMSAddress (void)) (void)
{
    _asm {
             mov     ax,4300h         //        int 2Fh with AX = 4300h
             int     2Fh              //        returns al = 80h
             cmp     al,80h           //        if an XMS server is present
             jnz     no_xms

             xor       bx,bx          //        faint pessimism!
             mov       es,bx
             mov     ax,4310h         // int 2Fh with AX = 4310h
             int     2Fh              // returns with es:bx addressing the
```

```
            mov     ax,bx           // calling point for the XMS functions
            mov     dx,es
            jmp     done

        no_xms:
            xor     ax,ax
            cwd
        done:
    }
    // return value in dx:ax
}
```

The various XMS functions provided by the server at the calling point returned in es:bx cover all the factors that impinge on real-mode programs seeking access to memory that DOS cannot handle by default—access to the HMA, control of the A20 line, access to extended memory, and access to UMBs, as shown in Table 2.

Table 2: Brief Description of XMS Functions

NUMBER	NAME
00h	Get XMS Version
01h	Allocate High Memory Area (HMA)
02h	Free HMA
03h	Global Enable A20 Line
04h	Global Disable A20 Line
05h	Local Enable A20 Line
06h	Local Disable A20 Line
07h	Query A20 Line Status
08h	Query Free Extended Memory
09h	Allocate Extended Memory Block (EMB)
0Ah	Free EMB
0Bh	Move EMB
0Ch	Lock EMB
0Dh	Unlock EMB
0Eh	Get Handle Information
0Fh	Resize EMB

Table 2: Continued

NUMBER	NAME	
10h	Allocate Upper Memory Block (UMB)	
11h	Free UMB	
12h	Resize UMB	(new to XMS 3.0)
88h	Query Free Extended Memory	(32-bit variation—new to XMS 3.0)
89h	Allocate Extended Memory Block (EMB)	(32-bit variation—new to XMS 3.0)
8Eh	Get Handle Information	(32-bit variation—new to XMS 3.0)
8Fh	Resize EMB	(32-bit variation—new to XMS 3.0)

The last five functions are available only from XMS servers that implement the latest version of the specification. Changes in the major version number of HIMEM.SYS reflect support for a new XMS version. This book is not concerned with the long vanished XMS 1.0, which had the appearance of a trial run and whose interface differs enough from the now extant XMS 2.0 to defeat compatibility. As an aside, note that this early specification used different int 2Fh functions (1600h and 1610h) to establish the presence of the memory manager and communicate the XMS calling point. XMS 2.0 is supported by all HIMEM.SYS versions 2.xx, most particularly the version 2.77 distributed with DOS 5 and on which much of this book's analysis is based. The new XMS 3.0 is implemented after a fashion by the HIMEM.SYS version 3.07 included with Windows 3.1. Windows does not (yet) use the new 32-bit variations which support more than 64MB of memory. In the author's opinion, the new XMS 3.0 functions are so badly implemented in HIMEM.SYS as to render their use unreliable, inasmuch as it is often feasible for a program to detect whether it is dealing with HIMEM.SYS or some other XMS server. Information to support this assertion is provided in Chapter 12, which examines the HIMEM.SYS implementation at close range.

Programs select an XMS function with register AH. Other information is exchanged in registers, typically in ax, bx, and dx (or the 32-bit forms eax, ebx and edx). The specification provides a comprehensive error-reporting scheme, so it is not necessary for an XMS server to implement all of these functions. HIMEM.SYS, for instance, does not support the three upper memory management functions (the very last of which is a recent addition to the XMS) but leaves them instead to whichever program actually provides the RAM (for example, EMM386.EXE).

As a quick example of an XMS call, consider the following short file (a fragment of a program developed over the several chapters of Part 3), which reports the results of a call to the Get XMS Version function. It is assumed that the calling address has already been found (see `GetXMSAddress` in GETXMS.C) and saved.

```
/* VERSION.C */

#include    <stdio.h>

#include    <standard.h>

extern void (_far *lpXMS) (void);

static BYTE ConvertBCD (BYTE);

void ReportVersion (void)
{
    WORD XMSversion, HIMEMversion;

    /*   Give the version numbers of both the XMS server (typically the
         device driver HIMEM.SYS) and the XMS specification to which the
         server adheres. Note that XMS function 00h returns the version
         numbers as packed BCD. */

    _asm {
              xor     ah,ah             //  AH = 00h is the function number;
              call    lpXMS             //  a far call to the XMS server;
              mov     XMSversion,ax     //  values are returned in registers
              mov     HIMEMversion,bx   //  ax and bx (also in dx, but not
    }                                   //  needed here)

    printf ("\nXMS driver version %hu.%02hu (Specification %hu.%02hu)\n",
         ConvertBCD (HIGHBYTE (HIMEMversion)),
         ConvertBCD (LOWBYTE (HIMEMversion)),
         ConvertBCD (HIGHBYTE (XMSversion)),
         ConvertBCD (LOWBYTE (XMSversion)));
}

static BYTE ConvertBCD (BYTE bcd)
{
    _asm {
              mov     al,bcd
              mov     ah,al
              mov     cl,04h
              shr     ah,cl
              and     al,0Fh
              aad                     // acts like al = ah * 10 + al, ah = 0
```

Two version numbers are provided—one for the XMS as a general interface and another for the specific implementation. Since the latter need not be HIMEM.SYS, it is impractical to rely on the driver version to identify the capabilities of an XMS server—except perhaps in a negative sense, to reject all drivers (from whatever source) whose version number matches that of a server known to have some failing.

A detailed description of the XMS functions—or rather, of HIMEM's particular representation of them—is presented after an examination of the important issues governing the interface design and its implementation as a service to real-mode programs.

CHAPTER 8

Extended Memory Access from Real Mode

To justify the development and study of the XMS, it must first be established that extended memory can be used by a real-mode program without degrading system performance, at least not so much as to compel migration to a full protected-mode operating system, nor even to an environment in which a large (but restricted) class of programs written for real mode are actually executed in protected mode. Readers who persevered with the sample device driver in Chapter 7 will have been forewarned that obtaining efficient access to extended memory for use by a real-mode program is not a trivial matter.

The obvious approach implemented by the ROM BIOS as int 15h function 87h (Move Extended Memory Block) is to step the processor up to protected mode, transfer data on behalf of the real-mode client, and then return to real mode. It was noted that this simple scheme has some difficulty accommodating hardware interrupts, since the corresponding service routines are almost certainly unprepared for protected mode. The BIOS function disables interrupts throughout its protected-mode operations—an approach that ensures both speed and integrity during the transfer but leaves the processor unresponsive to external influences for possibly long periods.

An alternative to ignoring hardware interrupts during protected-mode operations is to install handlers that "reflect" control to real mode so that the interrupt request may be serviced by the very code that was responsible before the transition to protected mode. (Interrupt reflection is basically an interrupt chain with a mode switch.) This keeps interested parties properly informed about external conditions but imposes a considerable overhead just to construct the protected-mode environment, at least some of which will have to be rebuilt after each reflection. Each interrupt will incur a significant time penalty, made even worse with the 80286 processor, for which stepping down to real mode is a tortuous operation whose frequent repetition is certainly unwelcome. All this may be worthwhile for DOS extenders, which construct a protected-mode environment in which well-behaved real mode code will still execute, but this is some distance from the universality sought here.

How then, is access to extended memory in protected mode to be reconciled with the need to preserve the real-mode treatment of hardware interrupts? The int 15h function attacks the problem by discounting the latter, but fortunately it is also possible to circumvent the requirement of protected mode. Attention need only be directed at the realities of processor design rather than formal concepts of memory addressing.

The Descriptor Cache

When the CPU decodes a memory reference it calculates an effective address from the base, index, and displacement to apply as an offset within the segment indicated by the instruction. To the CPU, this segment is described not by the value currently in the corresponding register, such as CS, DS, or ES, but by an internal representation—a sort of invisible component to the segment register—containing the segment's base, limit, and access rights. In protected mode, these properties are found by consulting a descriptor table in memory, using part of the segment register as an index. Real mode and the simulation known as virtual-8086 mode enforce a rigid relationship in which the segment base is obtained from the register by shifting four bits left. Either way, the processor need redraft its picture of a segment only when an instruction (such as LES BX or MOV ES, AX) loads the corresponding register and not with every memory reference. For this reason the internal representation is often referred to as a descriptor cache.

Software control of the descriptor cache is the key to accessing extended memory efficiently from real mode. Transition between operating modes does *not* flush the descriptor cache. Thus, it is possible to address extended memory from real mode by entering protected mode temporarily to set appropriate bases and limits for certain segments, knowing that after returning to real mode, the processor will continue with the descriptors that were loaded in protected mode. On the 80286, no direct support is offered for the reversion to real mode, but an undocumented processor instruction (LOADALL) can load the machine state from a structure in memory and allows the descriptor cache to be manipulated independently of the visible registers.

In the form presented so far, these possibilities amount not so much to an academic exercise as an entirely fruitless one, since interrupts could not be reenabled for a sequence of instructions that access extended memory without the risk that some interloper will save and restore the segment registers, acting quite properly but unwittingly corrupting the carefully prepared descriptor cache. Before any instruction intended to address extended memory is allowed to execute, some mechanism must be in place to detect whether an interrupt has changed the descriptor cache for the segment involved in the memory reference and, if so, to reload the "unreal" segment properties before proceeding.

This correction process should be invoked as rarely as possible. First priority should be given to noticing whether the relevant segment's properties have reverted to their real mode state, but this must be inferred indirectly, since the descriptor cache is not accessible to programmers. If this proves to be unfeasible, then some method must be found of detecting that an interrupt has occurred (and might therefore have altered the segment properties)—a tricky problem, given that interrupt handlers are supposed to act more or less invisibly.

The 80386 Flat-Memory Model

The 80386 offers the opportunity to achieve the primary goal through 32-bit addressing. Segment registers may be loaded in protected mode with values that select a segment extending to the end of the 80386 address space. Until these are reloaded in real mode, 32-bit instructions may address extended memory. Should an interrupt handler write to a segment register, causing a recalculation of the descriptor cache according to real-mode rules, then the next instruction intended to address extend memory may be affected in one of the following ways.

- If the limit field has been reduced (presumably to a 16-bit maximum), then the instruction may yield an effective address (offset) that is too large. This will cause the processor to raise an exception.
- On the other hand, if the limit is unchanged or if the offset happens to lie within the new limit, then the instruction will be allowed to proceed, but the address it accesses will be the intended one only if the new base calculated in real mode equals the base set in protected mode.

Provided that each selector loaded in protected mode has a real-mode-alias relationship to its segment's addressing base, such as selector 0008h indexing a segment with base 00000080h, the exception will act as the notification that the relevant segment's properties have changed. The code responsible for accessing extended memory need only install its own handler for the processor's real-mode version of the General Protection fault so that it may increase the segment size again.

Such contrivance is not necessary, however, for it happens that instructions that write to a segment register in real mode cause the processor to recalculate the segment base according to the real-mode rule of shift four bits left but *not* to impose the expected 16-bit limit of 65,535. This quirk means that a program may step up to protected mode temporarily to set a segment size of 4GB for real mode, knowing that *in the absence of any other protected mode software*, this 4GB limit will stay forever. It is important to understand that this real-mode version of the Flat Memory Model (sometimes called "Big Real Mode") is inherently unstable. In the protected-mode Flat-Memory Model, segment limits are under the control of the ring 0 supervisor and cannot be altered without consent, but in the real-mode version *any* process may jump to protected mode at will, probably returning with segment limits set to 65,535.

Finding the Segment Limit

It may be instructive to present some code to establish the existence of this quirk and to demonstrate in miniature some of the considerations that arise when manipulating segment limits as a technique to access extended memory. (It is worth noting that HIMEM.SYS accesses extended memory directly from real mode by using this technique.) The descriptor cache cannot be inspected directly by programmers, so finding the limit currently in force for a given segment register requires a trial-and-error determination of the largest effective address the processor will tolerate without generating an exception. While executing instructions that may violate the segment limit, an int 0Dh (GP fault) handler must be in place to field any possible exception.

The following routine presented in GETLIMIT.ASM is designed to be called in real mode. Even then conflict between the int 0Dh protection fault and hardware interrupt level 5 is possible if the Programmable Interrupt Controller (PIC) has been initialized as usual with int 08h as its base level (i.e., int 0Dh handles both IRQ 5 and the general-protection fault).

More problematic is the behavior of the routine when run in a virtual machine. Should the effective address be too high, the resulting processor exception will be gated to the virtual-8086 monitor in ring 0. There is a good chance of incurring a brutal response: Windows, for example, replies by destroying the virtual machine. If the routine were part of a larger program that ought not to be terminated pre-emptively, then it would be wise to prevent execution inside a virtual machine. At the other extreme, it is conceivable that in some situations the monitor may decide that system security is under no threat and will allow the virtual machine to handle its own faults, as if it really were running in real mode. One would not expect Windows to react this way since Windows has other uses for the memory outside any one virtual machine's megabyte, but the gentle reaction might be thought possible from memory managers, which exist only to enhance the original real-mode DOS environment and might therefore try to interfere as little possible.

As an illustration, consider the behavior of EMM386. On receiving the General Protection fault in ring 0, it inspects the offending instruction but decides that the ADDRSIZ: prefix used to generate a 32-bit address in a real-mode program's 16-bit segment is of no special interest and reflects the fault to the virtual machine. This is a very considerate, maybe even ambitious, attempt to reproduce the real mode environment. It may also just be carelessness, which seems likely, since EMM386 reflects the General Protection fault as int 06h (for Invalid Opcode) instead of the int 0Dh that the real-mode program will be expecting.

```
; GETLIMIT.ASM

;   This file contains a function that returns the 32-bit segment limit
;   currently applied to the ES register.

.MODEL                  SMALL, C
.386

;   ==================================================================

.CODE

GetESLimit                      PROC      PUBLIC USES ds
```

```
;   Install handlers for the real-mode Invalid Opcode and General Protection
;   faults. Store the current vector addresses in the code segment for
;   simplicity. Disable interrupts throughout the test, but note that the
;   caller must do this beforehand if the validity of the result on return
;   is to be ensured.

                        xor     ax,ax
                        mov     ds,ax

                        mov     ax,cs
                        shl     eax,10h
                        mov     ebx,eax

                        mov     ax,OFFSET int06h
                        mov     bx,OFFSET int0Dh

                        pushf
                        cli

                        xchg    eax,ds:[06h * 4]
                        xchg    ebx,ds:[0Dh * 4]

                        mov     cs:[lpOrgInt06h],eax
                        mov     cs:[lpOrgInt0Dh],ebx

;   Try reading a byte of memory to see if a processor fault is generated.
;   Adopt a bisection algorithm to determine the highest address that does
;   not trigger a fault:

;   ebx = least linear address known to be invalid (or 00000000h)
;   ecx = current testing address
;   edx = greatest linear address known to be valid

;   Thus in general, edx <= ecx < ebx.

                        xor     ebx,ebx
                        xor     edx,edx
                        xor     ecx,ecx
                        dec     ecx
read::
                        mov     al,es:[ecx]

;   If the address in ecx is valid (i.e., the previous instruction executes
;   normally), copy it to edx (highest known valid address) and continue.
```

```
                        mov     edx,ecx
                        jmp     @f

;   Attempting to read from an address regarded as invalid for segment ES
;   will generate an exception. The handler (given below) will redirect
;   control here, so that ebx (lowest invalid address) may be updated
;   instead of edx.

fault::
                        mov     ebx,ecx
@@:

;   Bisect the area of unknown behavior and reiterate.

                        mov     ecx,ebx
                        sub     ecx,edx
                        jb      done
                        shr     ecx,1
                        jz      done
                        add     ecx,edx
                        jmp     read
done:

;   Restore the original exception handlers and the caller's interrupt
;   state.
                        mov     eax,cs:[lpOrgInt06h]
                        mov     ds:[06h * 4],eax
                        mov     eax,cs:[lpOrgInt0Dh]
                        mov     ds:[0Dh * 4],eax
                        popf

;   Return the 32-bit limit (now in edx) as dx:ax for the convenience of
;   C language callers.

                        mov     ax,dx
                        shr     edx,10h
                        ret

GetESLimit              ENDP

;   ----------------------------------------------------------------------

int0Dh                  PROC    FAR PRIVATE

;   Check that the handler has not been invoked while a hardware interrupt
;    at the conflicting level (IRQ5) is pending.

                                push      ax
                                mov       al,0Bh    ;  signal the PIC to return
                                out       20h,al    ;  its In-Service register
                                jmp       @f        ;  on the next read
@@:
                                in        al,20h
                                test      al,01h SHL (0Dh - 08h)
                                pop       ax
                                jnz       chain

;    If the faulting instruction is the one expected to trigger a fault,
;    redirect control as described above.

                                call      RedirectFault
                                jnz       chain
                                iret
chain:
                                jmp       cs:[lpOrgInt0Dh]

int0Dh                          ENDP

;    ----------------------------------------------------------------------

int06h                          PROC      FAR PRIVATE

;    An Invalid Opcode handler is needed if running under a virtual machine
;    created by EMM386. It is the same as the int 0Dh handler above, but
;    without needing to resolve a possible conflict with IRQ5.

                                call      RedirectFault
                                jnz       chain
                                iret
chain:
                                jmp       cs:[lpOrgInt06h]

int06h                          ENDP

;    ----------------------------------------------------------------------

RedirectFault                   PROC      PRIVATE USES ax, retaddr:DWORD

;    If the faulting instruction is the one expected to trigger a fault,
;    change the interrupt handler's return address and return zf = ZR.
```

```
                              mov       ax,cs
                              cmp       ax,word ptr [retaddr + 02h]
                              jnz       @f
                              cmp       word ptr [retaddr],OFFSET read
                              jnz       @f

                              mov       word ptr [retaddr],OFFSET fault
@@:
                              ret

RedirectFault                 ENDP

;   ------------------------------------------------------------------

;   Storage for the int 06h and 0Dh address:

lpOrgInt06h                   dd        ?
lpOrgInt0Dh                   dd        ?

END
```

Changing the Segment Limit

To set a new segment limit requires a quick jump to protected mode, a barely functional example of which is presented in the next assembly language file. Again, the routine is designed to be run from real mode, which always allows the transition. Instructions such as LGDT, which prepare the protected mode environment, are sensitive to privilege level and attempting to execute this sort of code in a virtual machine is almost certain to meet an aggressive response from the operating system in ring 0. Windows, which regards the "DOS box" as an application program, destroys the machine as before. EMM386, however, is prepared to give way in these circumstances.

```
EMM386 Privileged operation error #00 -
Deactivate EMM386 and Continue (C) or reBoot (B) (C or B) ?
```

presumably conceding that from the perspective of the primary DOS session, EMM386 is the servant. Not all expanded memory managers take such a servile view. From a technical perspective it is certainly easier just to declare the problem as fatal. Continuation means that EMM386 will return the session to real mode so that the offending software may execute in the environment for which it appears to have been written. Some memory managers will simply not be able to do this, since the drive to provide more memory for the DOS session may have caused their designers to sacrifice the possibility of returning to any sort of sensible real-mode environment. Indeed, even though EMM386 is able in principle to disable itself and return to real mode, circumstances may prevent it from doing so. If it has been using the processor's paging facility to provide UMBs, then this is replaced by the more brutal message

```
EMM386: Unrecoverable privileged operation error #00 - press ENTER to reboot
```

To display this message, EMM386 reverts to real mode (automatically disabling paging), and, since the system is unlikely to survive the sudden loss of the device drivers and TSRs that users tend to place in UMBs, the likely consequence is a system crash. If any of the int 10h (video display) chain passes through a UMB, then the user won't even see the explanatory message. In fairness to Microsoft, it should be observed that this oversight has been remedied in the version of EMM386 distributed with Windows 3.1.

```
; SETLIMIT.ASM

;   This file presents a minimal routine that steps up to protected mode
;   temporarily just to load the invisible part of the ES register with some
;   given limit.

.MODEL                       SMALL, C
.386P

SEGMENT_DESCRIPTOR STRUCT
  limit                      dw        ?
  base                       dw        ?
  base_middle_8              db        ?
  access_rights              db        ?
  extra_byte                 db        ?
  base_high_8                db        ?
SEGMENT_DESCRIPTOR ENDS

TABLE_DESCRIPTOR   STRUCT
  limit                      dw        ?
  base                       dd        ?
TABLE_DESCRIPTOR   ENDS

;   ===================================================================

.DATA

;   The Global Descriptor Table used by the following routine provides just one
;   selector other than the null selector. Interrupts are disabled while
;   the procedure runs in protected mode and in the absence of processor
;   exceptions (or non-maskable interrupts, which the caller must disable).
;   No need exists for an Interrupt Descriptor Table, code segment, or stack
;   segment, but any serious application ought to provide a more
;   substantial, secure environment.

gdt                         SEGMENT_DESCRIPTOR <>,
                                               < , 0000h, 00h, 93h>

ES_selector                 =        SIZEOF SEGMENT_DESCRIPTOR
ES_descriptor               =        gdt + ES_selector

fpGDT                       TABLE_DESCRIPTOR   <SIZEOF gdt - 0001h>

;   ======================================================================

.CODE

SetESLimit                  PROC     PUBLIC USES es, limit:DWORD

;   Set the limit fields in the descriptor for the test segment. Note that
;   limits requiring more than the 20 bits available in the descriptor must
;   be stored as some multiple of the ubiquitous 4KB page. The granularity
;   bit (the MSB in the byte that also contains the four highest bits of the
;   limit) must be set to indicate that the limit has been scaled to fit.

                            xor      bl,bl
                            mov      eax,[limit]
                            test     eax,0FFF00000h
                            jz       @f
                            shr      eax,12
                            mov      bl,80h
@@:
                            mov      [ES_descriptor.limit],ax
                            shr      eax,10h
                            or       al,bl
                            mov      [ES_descriptor.extra_byte],al

;   Calculate the base address of the GDT and load the corresponding table
;   register. Make sure that interrupts are in fact disabled.

                            mov      ax,ds
                            movzx    eax,ax
                            shl      eax,4
                            add      eax,OFFSET gdt
                            mov      [fpGDT.base],eax
```

```
                         mov      bx,ES_selector

                         pushf
                         cli

                         lgdt     fword ptr [fpGDT]

;   Enter protected mode to load the ES register with the selector to the
;   prepared segment. Flush the instruction queue (with a JMP) to ensure
;   that the descriptor cache for ES is not loaded while the processor is
;   still in real mode.

                         mov      eax,cr0
                         or       al,01h
                         mov      cr0,eax
                         jmp      @f      ; flush instruction queue
@@:
                         mov      es,bx

;   Return to real mode and restore the caller's interrupt state.

                         and      al,NOT 01h
                         mov      cr0,eax
                         popf

;   Epilogue code restores the visible part of the ES register. Flush the
;   instruction queue beforehand to make sure that the processor is back in
;   real mode when it decodes the POP ES instruction. The processor
;   recalculates the base address, but does it also impose a 16-bit segment
;   size of 64K?
                         jmp      @f
@@:
                         ret

SetESLimit               ENDP

END
```

Now to make a program out of these assembly language procedures. The resulting SEGLIMIT gadget demonstrates the behavior of the descriptor cache for the ES segment across a transition from protected mode to real mode. Specifying a hexadecimal number on the SEGMENT command line allows an arbitrary segment size to be set for the ES register by taking a quick round trip to protected mode and then provides independent verification using a trial-and-error search. Alternatively, SEGLIMIT (with no command-line argument) may be used just to determine the current segment limit.

SEGLIMIT shields users from the effects of running the program in a virtual machine (see VM.C for how SEGLIMIT determines that it is running in a virtual machine, as opposed to genuine real mode). By default, the program refuses to execute in a virtual machine, since even attempting to discover the segment limit may trigger a protection violation or worse. SEGLIMIT does provide a /v switch so that users may override this and test their memory manager's generosity in tolerating a program's non-invasive testing of its environment. However, attempting to *change* segment limits will almost certainly be regarded as a privilege violation and is always disallowed.

```
/* MAIN.C */

#include    <dos.h>
#include    <stdarg.h>
#include    <stdio.h>
#include    <string.h>

#include    <standard.h>

static CHAR description [] = "\n\
Demonstrates behavior of the descriptor cache for the ES segment\n\
across a transition from protected mode to real mode.\n\
\n\
SEGLIMIT [limit] [/v]\n\
\n\
where the limit is interpreted as hexadecimal and for safety, may not\n\
be < FFFF.\n\
\n\
By default, the program does not execute in a virtual machine, since\n\
even attempting to discover the segment limit may trigger a protection\n\
violation (or worse). Use the /v switch to override this.\n\
";

extern DWORD GetESLimit (void);
extern void SetESLimit (DWORD);
extern BOOL InProtectedMode (void);
extern BOOL Is80386 (void);

static BOOL ReadLongHex (CHAR *, DWORD *);
static void PutResult (CHAR *, DWORD);
static void PutError (CHAR *, ...);

int main (int argc, char **argv)
{
    register CHAR *ptr;
    DWORD limit;
```

```
BOOL limit_given = FALSE;
BOOL vm_ok = FALSE;

/*    Parse the command line, accepting a /? switch for a program
      description or a hexadecimal limit for ES. For the /v switch,
      accept any initial substring of "VirtualMachineAllowed",
      independent of case. */

while (++ argv, -- argc) {
      ptr = *argv;
      if (*ptr == '/') {
          ptr ++;
          if (ptr [0] == '?' AND ptr [1] == '\0') {
                printf (description);
                return (1);
          }
          if (strnicmp ("VirtualMachineAllowed", ptr, strlen (ptr)) == 0) {
                vm_ok = TRUE;
                continue;
          }
          ptr --;
          PutError ("invalid switch - %s", ptr);
          return (-1);
      }
      else {
          if (limit_given) {
                PutError ("too many parameters");
                return (-1);
          }
          if (ReadLongHex (ptr, &limit) AND NOT limit < 0xFFFF) {
                limit_given = TRUE;
                continue;
          }
          PutError ("invalid parameter - %s", ptr);
          return (-1);
      }
}

/*    Establish that the 80386-specific functions for manipulating segment
      limits are usable. */

if (NOT Is80386 ()) {
      PutError ("requires 80386 CPU or better");
      return (-1);
}
```

```
    if (InProtectedMode ()) {
          if (NOT vm_ok OR limit_given) {
              PutError ("CPU is presently in protected mode");
              return (-1);
          }
    }

    /*    Disable interrupts for the test, which may possibly involve setting a
          new limit, then see if the operation was successful. Note that
          segment sizes below 64KB will not be welcomed when interrupts are
          reenabled. At present, the program disallows attempts to set
          segment limits < FFFFh, but an alternative would be to test whether
          or not small limits can be set, as long as a reasonable segment
          limit is restored before re-enabling interrupts. */

    _disable ();
    if (limit_given) SetESLimit (limit);
    limit = GetESLimit ();
    _enable ();

    printf ("\n%s %08lXh\n",
          limit_given ? "ES limit changed to" : "Current ES limit is", limit);
    return (0);
}

static BOOL ReadLongHex (register CHAR *ptr, DWORD *result)
{
    register WORD ch;
    DWORD temp = 0;

    ch = *ptr;
    if (ch == '0' OR ch == '\\') {
          if ((ptr [1] & ~0x20) == 'X') ptr += 2;
    }
    while (*ptr == '0') ptr ++;

    while (ch = *ptr ++) {
          if (ch >= '0' AND ch <= '9') ch -= '0';
          else {
              ch &= ~0x20;
              if (ch >= 'A' AND ch <= 'F') ch -= ('A' - 10);
              else if (ch == 'H') break;
              else return (FALSE);
          }
          if (temp & 0xF0000000) return (FALSE);
```

```
        temp <<= 4;
        temp += ch;
    }
    *result = temp;
    return (TRUE);
}

static void PutError (CHAR *str, ...)
{
    va_list va_start (argptr, str);
    printf ("\nSEGLIMIT fatal error:  ");
    vprintf (str, argptr);
    printf ("\n");
    va_end (argptr);
}
```

Virtual Machine Detection

Programs designed for real mode may need to establish that they are genuinely executing in real mode rather than a virtualization. The standard method on an 80386 processor is to inspect the Protection Enable (PE) bit of CR0—one of the processor's control registers. A DOS session run as a virtual machine (for example, a DOS box in Windows enhanced mode) is just a special protected-mode task in which the processor forms addresses by following 8086 rules rather than by indexing via descriptor tables. So, although the virtual machine runs programs written for real mode, inspection of the CR0 register will reveal that the processor is in fact in protected mode.

Some care is required however, because the obvious instruction

```
                mov      eax,cr0
```

to retrieve the control register into eax is in fact a privileged instruction in protected mode. The virtual-8086 task runs in ring 3, so the value loaded into eax by this instruction is entirely at the discretion of the monitor in ring 0. It is even conceivable that the instruction might not be permitted to execute. As an example, note that the exception handler in the VMM device of Windows enhanced mode simply steps over the instruction, leaving eax with whatever value it had before. EMM386 returns the true PE bit but hides the fact that it has enabled paging.

Thus, successful inspection of the PE bit can be guaranteed only if the 80286 instruction

```
                smsw     ax
```

is used, even if it is executing code that need only run on an 80386 or better. This instruction returns only the low word of CR0—corresponding to the Machine Status Word (MSW) of the 80286—but is not subject to privilege rules.

```
/* VM.C -- Compile with /G2 switch. */

int InProtectedMode (void)
{
    /*   Load ax with the PE bit of the processor's status word. Since this
         is bit 0, the result also serves as a Boolean return value. */
    _asm {
            smsw    ax
            and     ax,0001h
    }
}
```

CPU Identification

A function to identify the CPU as an 80386 will be needed again, hence its separation from the main source file. The standard test for processor differentiation—used in one form or another by HIMEM, EMM386, DEBUG, and the SYSINIT module of IO.SYS—checks the behavior of bits in the FLAGS register. Bits 12 to 15 are permanently set on the 8086 and 80186, but permanently clear on the 80286. For higher processors bit 15 is permanently clear but the other three of these bits are used for other purposes. At least some of them are certain to be set if the CPU is in virtual-8086 mode; in real mode, the bits will hold whatever values are entered into them.

On occasion deeper inspection may be warranted. The 16-bit DOS extender—DOSX.EXE—for Windows standard mode uses the following features for CPU identification.

- The shift count in an instruction such as SHL AX,CL is taken modulo 32 on the 80186 and all later processors, but the 8086 allows all 8 bits of CL to contribute.
- On both the 8086 and 80186, the PUSH SP instruction places the post-decrement value of SP on the stack. For later processors, the sequence PUSH SP followed by POP AX leaves AX and SP containing the same value (as it would for any other pair of registers).
- The SGDT instruction introduced for the 80286 processor stores a 24-bit base in its 6-byte operand, setting the last byte to FFh. If the instruction is used on later processors when the operand size attribute is 16-bit, then the last byte is cleared to zero.

The equivalent program for Windows enhanced mode—WIN386.EXE—uses the standard test to reject the 80286 and lower processors but is more concerned with differentiating the 80386 and 80486. The latter may be distinguished by its provision of an Alignment Check bit in the EFLAGS register, which the 80486 sets whenever it detects that the stack is aligned to a DWORD boundary. This method has the attraction of exploiting a clearly-defined advance in processor capabilities rather than relying on the behavior of otherwise unused flag bits; another difference, used by Compaq's version of HIMEM.SYS, is that the high byte of the machine status word (retrieved by the SMSW instruction) is fixed to FFh on the 80386.

Incidentally, Intel has belatedly recognized the need for processor identification; beginning with the Pentium processor, there is a CPUID instruction (opcode 0Fh A2h) that returns information on the processor, including its model and "stepping" (version).

As an aside on the problems that may befall a developer whose work depends too closely on processor behavior, note that WIN386 also guards against two 80386 defects. Some steppings of the 80386 have a problem with 32-bit multiplication, which WIN386 tries to uncover by multiplying 00000081h and 0417A000h repeatedly to see if the processor arrives at the correct answer reliably. A more serious problem is the existence of some 80386 chips that support an opcode sequence now used for the 80486 CMPXCHG instruction. WIN386 refuses to start if detects one of these chips, suggesting that the 80386 processor be upgraded or that Windows be run in standard mode. Exactly why this should be fatal is hard to assess without technical documentation from the likes of Intel or Microsoft, and it is all the more puzzling that the test appeared in versions of RAMDRIVE.SYS dated as early as May, 1988, when not many people can have known that CMPXCHG would appear in the 80486.

```
/* is80386.c */

int Is80386 (void)
{
    _asm {
            pushf               ; save flags - particularly the interrupt
                                ; flag, which is cleared during the test

            mov     ax,7000h            ; try setting the NT and IOPL bits
            push    ax                  ; but clearing the reserved bit F
            popf

            pushf
            pop     ax
            test    ax,8000h
            jnz     no                  ; if bit F comes on, then 8086 or 80186

            test    ax,7000h
            or        ax,ax
            jz        no                ; if other bits stay clear, then 80286
            mov     ax,0001h            ;  return TRUE for an 80386 or better
            jmp     done
        no:
            xor     ax,ax               ;  else return FALSE
        done:
            popf                        ;  restore original flags
    }
}
```

Instability of Flat Memory in Real Mode

Assemble, compile, and link to produce the executable SEGLIMIT.EXE, then prepare to experiment. Boot the computer with a minimal CONFIG.SYS to verify that real-mode segments are reported to be 64KB in size but may be enlarged to 4GB. For additional verification that SEGLIMIT's trial-and-error determination of segment limit works correctly, note that when setting large segment sizes, SEGLIMIT may report a slightly higher limit than it was directed to set. For example:

```
P:\XMS\ACCESS\SEGLIMIT>seglimit 12345678

ES segment limit changed to 12345FFFh
```

This is an expected consequence of page granularity in which the least significant 12 bits of a large limit must be scaled away when entering the limit in a segment descriptor, there being only 20 bits available. When the descriptor cache is refreshed from a descriptor that has a page granular limit, the processor reconstitutes a 32-bit value in which the lowest 12 bits are all set.

Installing HIMEM.SYS but no software that uses extended memory or enters protected mode produces the same situation—64KB segments that can be enlarged permanently to 4GB. Now install HIMEM and some program such as SMARTDRV.SYS (without its /a switch) or RAMDRIVE.SYS (with its /e switch) that will certainly have moved data to extended memory before the opportunity arises to run SEGLIMIT. The 4GB limit is already in place, demonstrating that HIMEM does indeed access extended memory directly from real mode by using the technique described in this section.

The instability of this Flat-Memory Model may be seen by booting with HIMEM and RAMDRIVE. Run some protected-mode software such as Windows, and, on return, segments will probably be back to 64KB in size. After copying a file to the RAM disk, the segment limit will again be FFFFFFFFh. Note that this effect might not be seen if SMARTDRV is installed. SEGLIMIT may seem to always report 4GB segments because by the time SEGLIMIT begins its investigation, the SMARTDRV cache in extended memory will have been used already (to get the SEGLIMIT.EXE file ready for execution).

The 80286 LOADALL Instruction

In the scheme outlined at first for the 80386, base addresses could be prepared in protected mode to have real-mode values so they would not be affected by an interruption that saved and restored the segment registers. The key was not the manipulation of base addresses but rather the use of segment limits large enough to reach into extended memory. Were the segment limit to change during an interruption (denying access to extended memory), the processor itself could be relied on to sound the alert via a General Protection fault. In fact it turns out that the processor does not alter segment limits in real mode, so base addresses can be removed from the scheme altogether.

None of this is possible on the 80286, which restricts segments to 64K. Discounting the HMA, extended memory cannot be addressed by a segment whose base lies in the first megabyte. Real-mode code may access extended memory only by using segments whose base addresses differ wildly from the normal real-mode values. An interruption that reloaded segment registers would cause recalculation of the base addresses according to the real-mode formula, and instructions that had been working with extended memory immediately beforehand would suddenly be addressing memory in the first megabyte. To the processor, the instructions remain valid, so there is no way to secure notification that the change has definitely occurred. Therefore, *all* interrupts must be prevented from returning to the instruction stream without first ensuring that the base addresses prepared for extended memory access are in place.

One feature common to all interrupts, irrespective of the code used to service them, is that the processor saves the CS register on the stack and restores it when the interrupt returns. In the case at hand, the protected-mode arrangements for extended memory access may also prepare a base address in the descriptor cache for CS which is not related to the visible CS register by the usual four-bit shift. The plan is that the instructions that access extended memory should execute with a false value for the visible CS. If an interrupt occurs, then the first instruction executed after the interrupt has finished will be fetched with respect to a segment base calculated from the restored CS register. It will not be the instruction that would have come next had the interrupt not occurred.

In this scheme, then, the series of instructions that work with extended memory by manipulating segment bases must be shadowed by corrective code addressable from another segment, just as instructions that rely on large segment limits must be protected by an int 0Dh handler. A workable implementation is easily sketched:

```
.MODEL                  SMALL, C

;   Code that works with extended memory is placed in the XM_TEXT segment.
;   The assembler builds a second segment SHADOW_TEXT that deals with
;   interrupts diverted automatically from the XM_TEXT segment. Offsets in
;   the two segments must correspond exactly, since the diversion is caused
;   by the change in segment base recorded in the processor's descriptor
;   cache. It helps to have both segments begin with a zero offset—
;   enforced by the following definitions.

XM_TEXT                     SEGMENT PARA PUBLIC 'CODE'
XM_TEXT                     ENDS
SHADOW_TEXT                 SEGMENT PARA PUBLIC 'CODE'
SHADOW_TEXT                 ENDS

;   ==========================================================================

.CODE                       XM_TEXT
begin_XM_TEXT               LABEL     BYTE

;   The code that sets up the descriptor cache need not be shadowed. It is
;   provided here as an external reference, necessarily in the XM_TEXT
;   segment to avoid reloading CS on return. It must be linked after all
;   modules that use shadow code.

                            EXTERN    LoadDescriptorCache:NEAR

xm_function                 PROC      FAR PUBLIC

;   Unfortunately, the assembler (MASM 6.0) determines incorrect locations
;   (without any form of warning) for its emissions in the SHADOW_TEXT
;   segment, unless steps are taken in advance to remove all doubt about
;   forward references. The following ruse with the ORG directive tells
;   MASM how much space to allow for the previous instruction—something it
;   ought to be able to figure out correctly.

@@:                         mov       bx,OFFSET xm_start
                            ORG       @b + 3
reset_xm::
                            push      bx

;   Call the procedure LoadDescriptorTable to set base addresses for
;   extended memory access (depending on the particular purpose in mind).
;   All general registers should be preserved and the procedure must return
;   with interrupts disabled. To support interrupt tolerance, the procedure
;   must also prepare the CS descriptor cache and visible CS register:

;           cs = SHADOW_TEXT    cs.base = XM_TEXT SHL 04h

;   Because the base address refers to XM_TEXT, execution will continue as
;   usual when the procedure has finished. Interrupts (once they are
;   enabled) will be diverted automatically to the SHADOW_TEXT segment.
```

```
                              call      LoadDescriptorCache

@@:                           mov       bx,OFFSET xm_done
                              ORG       @b + 3

                              sti
xm_interruptable::

;   Skip ahead to the instruction whose address was in BX at the label
;   reset_xm.

                              retn
xm_start:

;   Code for extended memory access—relying on the preparation of base
;   addresses independently of the visible segment registers—goes here.

;   Register BX has been reserved to keep track of interruptions and must
;   not be changed by any instruction in this section. Near jumps and
;   calls are allowed as long as their targets are also in this section.

;                             .
;                             .
;                             .

xm_done::
                              retf

xm_function                   ENDP

;   ====================================================================

.CODE                         SHADOW_TEXT
begin_SHADOW_TEXT  LABEL      BYTE

;   Advance the assembler's location counter to the same offset within
;   SHADOW_TEXT that the label xm_interruptable has within XM_TEXT.

REPEAT                        OFFSET xm_interruptable - OFFSET begin_XM_TEXT
                              nop
ENDM

;   Any interrupt occurring during the execution of code between the labels
;   xm_interruptable and xm_done in XM_TEXT will be diverted to the
;   corresponding offset within this segment. The DEC BX instructions
;   placed here count backward from the label xm_done (loaded into BX
;   before entering the interruptable section) to obtain the address of the
;   instruction to which control was meant to return.

REPEAT                       OFFSET xm_done - OFFSET xm_interruptable
                             dec      bx
ENDM

;   Return control to the instruction now addressed by BX in the XM_TEXT
;   segment, but only after resetting the base addresses for extended
;   memory access.
                             jmp      XM_TEXT:reset_xm

;   =========================================================================

;   Ensure that the two segments have equal size by padding the smaller one.
;   This will maintain the correspondence between addresses in the two
;   segments even if several modules of this form are linked together (in
;   which case, incidentally, the paragraph-alignment at the start of the
;   file will be needed in the first module only).

.CODE                        XM_TEXT
size_XM_TEXT                 =        $ - OFFSET begin_XM_TEXT

.CODE                        SHADOW_TEXT
size_SHADOW_TEXT   =         $ - OFFSET begin_SHADOW_TEXT

IF                           size_XM_TEXT LT size_SHADOW_TEXT
  .CODE            XM_TEXT
  REPEAT           size_SHADOW_TEXT - size_XM_TEXT
                             nop
  ENDM
ELSE
  .CODE                      SHADOW_TEXT
  REPEAT           size_XM_TEXT - size_SHADOW_TEXT
                             nop
  ENDM
ENDIF
```

Thus, in principle, the processor can be made to divert interrupts automatically—avoiding the unwieldy and system-dependent alternative of trapping all possible interrupts to redirect control when the original handlers try to return. Even so, there are still problems in practice if the only way to load base addresses independently of the visible segment registers is to step up to protected mode to load the visible register with a selector to a descriptor with the desired base address. The selector loaded into CS must be the real-mode segment of the shadow code to which interrupts get diverted, but this value will almost certainly be too large for a reasonable GDT. The range of allowable selectors is further constrained by the need for ring 0 privilege to get back to real mode.

As it happens, returning an 80286 to real mode is a lengthy ordeal in which the processor is halted and then reawakened by some device such as the keyboard controller. Because of this, any practical scheme for extended memory access on an 80286 must be able to load the descriptor cache without a transition to protected mode. The preceding discussion shows that even if reversion to real mode were an easy matter for the 80286, tolerance of interrupts would demand a better method of accessing the descriptor cache.

The expensive transition from protected mode to real mode on the 80286 was overcome in early versions of RAMDRIVE.SYS by using an undocumented CPU instruction that offers direct write-only access to the processor's internal register representations, taking values from the 66h-byte structure presented in Table 8-1. The instruction is known as LOADALL, although it is not recognized as such by many assemblers (but is accepted by the CodeView debugger) and must usually be specified through its opcode, 0Fh 05h.

Table 8-1a: Machine State Structure Expected at Linear Address 000800h by the 80286 LOADALL instruction

OFFSET	SIZE	DESCRIPTION
00h	3 words	?
06h	word	MSW
08h	7 words	?
16h	word	TR
18h	word	FLAGS
1Ah	word	IP
1Ch	word	LDTR
1Eh	word	DS
20h	word	SS
22h	word	CS
24h	word	ES
26h	word	DI
28h	word	SI
2Ah	word	BP

Table 8-1a: Continued

OFFSET	SIZE	DESCRIPTION
2Ch	word	SP
2Eh	word	BX
30h	word	DX
32h	word	CX
34h	word	AX
36h	3 words	ES descriptor cache
3Ch	3 words	CS descriptor cache
42h	3 words	SS descriptor cache
48h	3 words	DS descriptor cache
4Eh	3 words	GDTR descriptor cache
54h	3 words	LDTR descriptor cache
5Ah	3 words	IDTR descriptor cache
60h	3 words	TR descriptor cache

Table 8-1b: Descriptor Cache Format for the 80286 Machine-State Structure

OFFSET	SIZE	DESCRIPTION
00h	3 bytes	base address
03h	byte	access rights
04h	word	limit

At the programming level, LOADALL acts much like a multiple MOV instruction—intersegment JMP, LGDT, LIDT, LLDT, LMSW, LTR, and POPF all rolled into one—with the extra ability to write directly to the normally invisible descriptor cache. The instruction may be executed in real mode (it requires ring 0 privilege in protected mode), so base addresses can be prepared for extended memory without incurring the transition penalty. The scheme for enabling interrupts is supported by the complete independence allowed in the machine-state structure for the visible and invisible components of CS.

A slight problem is presented by LOADALL's insistence that the machine-state structure be located at the linear address 000800h—in memory occupied by the IO.SYS kernel file. In general, code that loads the descriptor cache using LOADALL must save and restore the 66h bytes beginning at that address, keeping interrupts disabled throughout. DOS versions 3.30 and 4.01 left the necessary space clear to spare RAMDRIVE this extra trouble. It is perhaps a measure of the fading significance of the 80286 that DOS 5 does not.

The 80386 LOADALL Instruction

If only for completeness, it should be observed that a facility to load the CPU state is also offered by the 80386. However, it is hard to think of a practical use for this in real mode, given the efficiency that attends the use of large segments.

The existence of an undocumented 80386 instruction probably related to the 80286 LOADALL may be inferred from the sort of close inspection of EMM386 needed when some code that works correctly in real mode crashes the system if run under EMM386's real-mode emulation. As noted earlier, Invalid Opcode and General Protection faults raised during the execution of a VM task are trapped in ring 0 by EMM386, which examines the offending instruction and reflects uninteresting cases to the virtual machine. Like most instructions, the opcode sequences 0Fh 05h and 0Fh 07h are reflected, but these two are first passed to pieces of stub code, lying side by side.

This special, though inconsequential, treatment may prompt one to check whether or not the opcodes have any significance to other software. The sophisticated debugger WDEB386 (distributed with the Windows SDK and acting much like a 386 version of the excellent SYMDEB, except for the unfortunate and surely unnecessary restriction to debugging through the serial port) disassembles 0Fh 05h as LOADALL and 0Fh 07h as LOADALLB BYTE PTR ES:[DI]. Analysis of this debugger's opcode tables confirms that the 'B' is appended to 'LOADALL' because, just as with SCASB and STOSB, the instruction's operand is the memory addressed by ES:DI or ES:EDI (depending on the current address size attribute), with no segment override permitted.

The format in which the instruction expects its input is widely known among BIOS designers. The 80386 does not recognize the original LOADALL and reacts to its use by raising an Invalid Opcode exception. This is unlikely to present a problem nowadays, but as late as July 1987, RAMDRIVE.SYS (version 1.19) still relied entirely on the 80286 LOADALL for its extended memory access—whatever the CPU. On many 80386 machines the BIOS counteracts the non-existence of the 80286 LOADALL by including an emulation facility in the default handler for the Invalid Opcode fault. In some cases emulation is achieved by jumping to protected mode, but in others the input intended for the 80286 LOADALL is recast for use by the 80386 equivalent.

Table 8-2a: Machine State Structure Read from ES:DI by the 80386 LOADALL Instruction

Items marked * are extended from 16 bits

Offset	Size	Description
00h	dword	CR0

Table 8-2a: Continued

OFFSET	SIZE	DESCRIPTION
04h	dword	EFLAGS
08h	dword	EIP
0Ch	dword	EDI
10h	dword	ESI
14h	dword	EBP
18h	dword	ESP
1Ch	dword	EBX
20h	dword	EDX
24h	dword	ECX
28h	dword	EAX
2Ch	dword	DR6
30h	dword	DR7
34h	dword	TR *
38h	dword	LDTR *
3Ch	dword	GS *
40h	dword	FS *
44h	dword	DS *
48h	dword	SS *
4Ch	dword	CS *
50h	dword	ES *
54h	3 dwords	TR descriptor cache
60h	3 dwords	IDTR descriptor cache
6Ch	3 dwords	GDTR descriptor cache
78h	3 dwords	LDTR descriptor cache
84h	3 dwords	GS descriptor cache
90h	3 dwords	FS descriptor cache
9Ch	3 dwords	DS descriptor cache
A8h	3 dwords	SS descriptor cache
B4h	3 dwords	CS descriptor cache
C0h	3 dwords	ES descriptor cache

Table 8-2b: Descriptor Cache Format for the 80386 Machine State Structure

OFFSET	SIZE	DESCRIPTION
00h	byte	?
01h	byte	access rights
02h	2 bytes	?
04h	dword	base address
08h	dword	limit

For a simple demonstration of how the LOADALL instructions work, without having to write a whole new program, consider a replacement for the SETLIMIT.ASM module used earlier in the SEGLIMIT program. Instead of a brief step up to protected mode to have the processor load the ES descriptor cache simply as its normal response to an instruction that loads the visible register, this alternative SETLIMIT.ASM uses the 80386 LOADALL instruction to enter values in the descriptor cache directly.

```
; SETLIMIT.ASM

.MODEL                          SMALL, C
.386P

;   Descriptor cache format for the 80386 machine state structure:

CACHE                           STRUCT
  unknown_00h                   db        ?
  access_rights                 db        ?
  unknown_02h                   db        ?
  unknown_03h                   db        ?
  base                          dd        ?
  limit                         dd        ?
CACHE                           ENDS

;   ========================================================================

.DATA

;   Statically allocated machine state structure for the 80386 LOADALL
;   instruction—observe that all segments are given access rights of 92h:

loadall_table                   LABEL     BYTE
```

```
loadall_cr0                     dd      ?
loadall_eflags                  dd      ?
loadall_eip                     dd      OFFSET loaded
loadall_edi                     dd      ?
loadall_esi                     dd      ?
loadall_ebp                     dd      ?
loadall_esp                     dd      ?
loadall_ebx                     dd      ?
loadall_edx                     dd      ?
loadall_ecx                     dd      ?
loadall_eax                     dd      ?
loadall_dr6                     dd      ?
loadall_dr7                     dd      ?
loadall_tr                      dd      ?
loadall_ldtr                    dd      ?
loadall_gs                      dd      ?
loadall_fs                      dd      ?
loadall_ds                      dd      ?
loadall_ss                      dd      ?
loadall_cs                      dd      ?
loadall_es                      dd      ?
loadall_tr_cache   CACHE        <>
loadall_idtr_cache CACHE        <>
loadall_gdtr_cache CACHE        <>
loadall_ldtr_cache CACHE        <>
loadall_gs_cache   CACHE        <00h, 92h, 00h, 00h, 00000000h, 0000FFFFh>
loadall_fs_cache   CACHE        <00h, 92h, 00h, 00h, 00000000h, 0000FFFFh>
loadall_ds_cache   CACHE        <00h, 92h, 00h, 00h, 00000000h, 0000FFFFh>
loadall_ss_cache   CACHE        <00h, 92h, 00h, 00h, 00000000h, 0000FFFFh>
loadall_cs_cache   CACHE        <00h, 92h, 00h, 00h, 00000000h, 0000FFFFh>
loadall_es_cache   CACHE        <00h, 92h, 00h, 00h, 00000000h, 0000FFFFh>

;   =================================================================

.CODE

SetESLimit          PROC        PUBLIC USES gs fs es ds edi esi, limit:DWORD
                                LOCAL    scratch:FWORD

;   Preserve registers important to system control (and whose values can be
;   determined in real mode)—the GDTR and IDTR table registers, CR0, and
;   the two debugging registers DR6 and DR7.

                                sgdt     [scratch]
                                mov     ax,word ptr [scratch]
                                mov     word ptr [loadall_gdtr_cache.limit],ax
                                mov     eax,dword ptr [scratch + 02h]
                                mov     [loadall_gdtr_cache.base],eax

                                sidt    [scratch]
                                mov     ax,word ptr [scratch]
                                mov     word ptr [loadall_idtr_cache.limit],ax
                                mov     eax,dword ptr [scratch + 02h]
                                mov     [loadall_idtr_cache.base],eax

                                mov     eax,cr0
                                mov     [loadall_cr0],eax
                                mov     eax,dr6
                                mov     [loadall_dr6],eax
                                mov     eax,dr7
                                mov     [loadall_dr7],eax

                                pushfd
                                pop     [loadall_eflags]

;       Make sure that all registers needed for access to the stack will retain
;       their values. Visible and invisible components of the SS register must
;       be prepared. Note that a 64KB limit for SS has been set statically.

                                mov     [loadall_ebp],ebp
                                mov     [loadall_esp],esp

                                mov     ax,ss
                                mov     word ptr [loadall_ss],ax
                                movzx   eax,ax
                                shl     eax,4
                                mov     [loadall_ss_cache.base],eax

;       Similarly, make sure that CS:IP will address the instruction immediately
;       after LOADALL. The value for the IP register is set statically.

                                mov     ax,cs
                                mov     word ptr [loadall_cs],ax
                                movzx   eax,ax
                                shl     eax,4
                                mov     [loadall_cs_cache.base],eax

;       Assign the designated segment limit to the ES descriptor cache.
```

```
                                mov     eax,[limit]
                                mov     [loadall_es_cache.limit],eax

;   The LOADALL instruction expects the machine state structure at ES:DI.
;   Immediately after the LOADALL instruction, all registers (including the
;   invisible components of segment registers) have values taken from the
;   machine state structure.

                                push    ds
                                pop     es
                                mov     di,OFFSET loadall_table

                                db      0Fh, 07h
loaded::

;   Epilogue code restores the visible components of the segment registers,
;   forcing a recalculation of base addresses in the descriptor cache but
;   leaving the new limit for ES in place.

                                jmp     @f
@@:
                                ret

SetESLimit                      ENDP

END
```

The SEGLIMIT program behaves in much the same way no matter which version of SETLIMIT.ASM is linked. The version which goes to protected mode and back to load the segment limit is faster, and, since efficient access to extended memory from real mode on an 80386 needs nothing else at CPU level other than the ability to set the limit field in the invisible part of a segment register, it is difficult to see how the 80386 LOADALL instruction can have much practical use.

One small difference between the two versions of SEGLIMIT arises because the 20-bit storage restriction on the limit field in a segment descriptor does not affect the LOADALL version, which manipulates the descriptor cache directly and need not lose the 12 least significant bits of the limit. Thus, comparison of

```
P:\XMS\ACCESS\LOADALL>seglimit 12345678

ES segment limit changed to 12345678h
```

with the result 1234FFFh obtained using the earlier version of SETLIMIT.ASM shows that the processor does indeed carry a 32-bit representation of the limit in the invisible part of a segment register, rather than attach some special significance to page granular segments.

Another difference can be seen if the SEGLIMIT program is changed to allow limits below FFFFh. This would not normally be sound, for it is unlikely that the real-mode operating environment is prepared for exceptions raised by the processor if the current instruction's operand extends past the limit for its segment. (Note that there really is such a thing as GP fault in real mode, e.g., for `mov ax, word ptr [0FFFFh]`.) Although the LOADALL version of SETLIMIT.ASM can set an arbitrarily small limit for ES, it can never restore a limit that has been set too low to contain the machine state structure (necessarily addressed with respect to ES).

Adventurous readers who care for the experiment may establish the curious property that even though the CPU uses just CCh bytes for the machine state structure, 0127h bytes must be available to prevent a General Protection fault. (Frank van Gilluwe's *The Undocumented PC* reports that the dword at offset 0CCh, immediately following the ES descriptor cache field in Table 8-2a, holds the length of the table and that the optional fields following this length refer to temporary registers.)

CHAPTER 9

The A20 Line

That efficient access to extended memory can be obtained for real-mode programs has now been demonstrated—but only from the processor's perspective. Most PC-compatible machines will in fact have been fitted with some mechanism to emulate the baseline 8086, by disabling one or more address lines beginning with the A20. Even after coaxing the CPU to address extended memory while still in real mode, there may also be a problem of physical access because the address decoded by the memory hardware is not the address generated by the processor.

On the 8086, which can form addresses with 20 bits only, an offset of 0010h applied to a base address of FFFF0h wraps around to the bottom of memory to location 00000h. The 80286 and more advanced processors provide more lines for addressing memory, so the same offset applied to the same base generates the linear address 100000h with no wrap-around. This means, of course, that just under 64KB of extended memory—the High Memory Area (HMA)—can be addressed directly from real mode by using segment FFFFh, but it also exposes a possible conflict with software written for the 8086. Should this be a problem, compatibility of the real-mode environment with the 8086 address space may be enforced by holding the A20 address line low, so that the memory hardware sees only the first 20 bits of any 21-bit address formed by the processor. This arrangement is actually adopted as standard for the real-mode initialization of machines intended primarily for running DOS applications. It seems that an appreciable volume of DOS software assumes the wrap-around at one megabyte.

It may be worth a small digression to allay any rising incredulity on this point. Readers who wonder how anyone can have produced code relying on this wrap-around and who are beginning to think that all this trouble stems from some obsessive desire of DOS's designers to support CP/M may do well to look at their own programs, for although readers are unlikely to have written such code, it turns out that the prime culprits are the natural allies of every programmer who thinks his executables are too large and load too slowly. Utilities such as the EXEPACK program, which used to accompany Microsoft's programming languages but is now incorporated as a LINK option, compress a program's code and data (and perhaps also the EXE header's relocation table) and then bind the result with some code that unscrambles the executable in memory at run time. Decompression necessarily works from the top of memory downward, perhaps requiring several cycles due to segmentation. Unpacking code in the first 64KB of memory requires a little care to make sure that the cycle does not start with an offset so high as to demand a segment below zero and thus rely (wittingly or not) on wrap-around to produce the desired decompression.

Physical deactivation of the A20—however necessary while operating in real mode—may have a rather brutal effect across the system. Depending on the A20 hardware, it is possible that addresses in the HMA are not the only ones to be altered on their way from the CPU to the memory unit. The A20 must be enabled for access to extended memory—even that which lies above the area of possible conflict with 8086 software.

Unfortunately, the mechanism for controlling the A20 line may vary considerably from one computer to another. Assistance from the ROM BIOS is available if the intention is to leave real mode and never return; int 15h function 89h (Enter Proctected Mode) enables the A20 and switches the processor to protected mode, creating a simple environment that the caller can adapt for grander purposes. Similarly, the A20 is enabled for the data transfer that int 15h function 87h conducts in protected mode on behalf of its real-mode caller, but this function disables the A20 before returning. Only rarely (if ever) is software support offered in the form of separate BIOS functions to enable and disable the A20.

In practice then, an XMS server designed to support programs running in real mode must have some knowledge (whether intrinsic or configurable) of the possible variety of A20 hardware. The server will need an A20 facility for itself—as part of the function it offers for extended memory access—but clients also may need to manipulate the A20 on their own, for access to the HMA or on either side of a transition to protected mode. To spare clients any concern about the underlying hardware, the XMS devotes a set of functions to A20 control.

If the XMS server is part of a 80386 or higher system running real-mode software in a virtual machine, the effect of the A20 can be emulated by paging the 64KB at address zero to the 1MB boundary (making a second copy) when the virtual A20 is disabled. To some extent, knowledge of the A20 hardware in this situation is less pressing, as long as programs in the virtual machine attempt A20 control only through the XMS interface. Indeed, a memory manager that intends never to return to real mode can get to protected mode with no knowledge of the machine's A20 hardware by using int 15h function 89h. However, bullet-proof virtualization still demands that the virtual-8086 monitor know about the hardware so that it can trap access to the relevant I/O ports and divert the A20 request to its paging routines.

Local, Global, and Physical

The XMS functions for A20 control are not provided as a simple trio of enable, disable, and query. Instead, an attempt is made at logical management, particularly so that code requiring an active A20 may ask for local enabling at the start and local disabling at the end, without troubling itself over the actual state of the A20 before and after. Thus,

```
LocalEnable ();
/* Insert program code that relies on the A20 being enabled */
LocalDisable ();
```

enables the A20 for code that accesses extended memory and restores the physical A20 state when the extended memory operations are complete.

The following pseudocode shows how an XMS server armed with private functions for physical control of the A20 and an internal counter might implement local management:

```
LocalEnable ()
{
    if (PhysicalQuery () == OFF) PhysicalEnable ();
    local ++;
}

LocalDisable ()
{
    /*  local == 0 at entry is an error condition  */
    if (local == 1 AND PhysicalQuery () == ON) PhysicalDisable ();
    else if (local > 1 AND PhysicalQuery () == OFF) PhysicalEnable ();
    local --;
}
```

Coded this way, routines for physical A20 control are called only when strictly necessary. This is crucial because A20 control can be very slow. Moreover, the local A20 functions not only preserve the relationship

```
PhysicalA20State = (local == 0 ? OFF : ON);
```

by which the server can match disables against enables, but also enforce it—allowing recovery from any direct hardware access that may have interfered with the server's bookkeeping. Reconciling the physical A20 state to the XMS server's internal record keeps everything in order for XMS clients but acts to the detriment of code that bypasses the XMS facilities for A20 control. One such piece of code is the BIOS implementation of int 15h function 87h, which disables the A20 after performing the requested data transfer; it is imperative that an XMS server designed for real-mode operation trap int 15h function 87h to preserve the physical state of the A20 line across the BIOS call.

Integration of local A20 management with physical control is offered through the global functions

```
BOOL GlobalEnable ()
{
    if (global == OFF) {
          LocalEnable ();
          global = ON;
    }
}

BOOL GlobalDisable ()
{
    if (global == ON) {
          LocalDisable ();
          global = OFF;
    }
}
```

which are apparently intended for use by the owner of the HMA. Even the `GlobalDisable` function will not actually disable the A20 if any `LocalEnable` calls are still pending.

The following A20MAP program combines the XMS functions for A20 control with one of the previous chapter's techniques for accessing extended memory. After marking each megabyte boundary in the CPU's main memory with a unique identifier, the A20 line is disabled. Comparison of the identifiers retrieved from each boundary with those known to have been written there reveals how deactivation of the A20 affects extended memory generally. (A20MAP saves and restores the memory affected by this testing.)

The A20 functions require only the function number in AH, and, although they may return an error code in register BL, it suffices here merely to know whether or not the XMS request has been successful—as indicated by the Boolean value returned in AX. This suggests implementation as C functions with in-line assembler, for example

```
BOOL LocalEnableA20 (void)
{
    _asm {
                mov     ah,05h
                call    lpXMS
    }
    // return value in AX
}
```

where `lpXMS` contains the address of the XMS calling point returned by int 2Fh function 4310h.

Readers should also be aware of the opportunity to generate in-line function calls whenever arguments are expected in the same registers (AX, DX and BX) used for the **_fastcall** convention, by which Microsoft C allows parameter-passing in registers rather than on the stack:

```
#define LocalEnableA20()    ((BOOL (_fastcall _far *) (WORD)) lpXMS) (0x0500)
```

The `_fastcall` attribute directs the compiler to place the first 16-bit argument in AX, the second (if present) in DX, and the third in BX—just as needed for many XMS calls.

The program's main procedure shows how code requiring an active A20 need simply be enclosed in calls to the `LocalEnableA20` and `LocalDisableA20` functions. Testing (conducted in a separate function) involves the physical deactivation of the A20, done by calling the `LocalDisableA20` function repeatedly and taking care to restore the XMS server's local state when the time comes to reenable the A20.

```
/* A20MAP.C */

#include    <dos.h>
#include    <stdarg.h>
#include    <stdio.h>
#include    <stdlib.h>

#include    <standard.h>
#include    <sysvars.h>

static CHAR description [] = "\n\
Determines the extended memory map when the A20 line is disabled.\n\
The present implementation has a granularity of one megabyte and\n\
requires an 80386 processor or better, operating in real mode.\n\
";

/*  Use XMS facilities for A20 manipulation. */

extern void (_far *GetXMSAddress (void)) (void);

static void (_far *lpXMS) (void);

#define LocalEnableA20()     ((BOOL (_fastcall _far *) (WORD)) lpXMS) (0x0500)
#define LocalDisableA20()    ((BOOL (_fastcall _far *) (WORD)) lpXMS) (0x0600)
#define QueryA20()           ((BOOL (_fastcall _far *) (WORD)) lpXMS) (0x0700)

static WORD PhysicalDisableA20 (void);

/*  Use the real-mode flat model to form addresses with more than 20 bits
    independent of any interface that enables the A20. Unfortunately,
    this limits the program to use on an 80386 or better. */

extern DWORD GetESLimit (void);
extern BOOL InProtectedMode (void);
extern BOOL Is80386 (void);
extern void SetESLimit (DWORD);
```

```
/*  Functions and variables for reading and writing test words at successive
    megabyte boudaries  */

static WORD GetTotalMemory (void);
extern WORD ReadID (WORD);
static CHAR *TestMemoryMap (WORD);
extern void WriteID (WORD, WORD);

#define  CAPACITY 16
static WORD disabled [CAPACITY], originals [CAPACITY];

/*  Message output  */

static void PutError (CHAR *, ...);

/*  ==================================================================  */

int main (int argc, char **argv)
{
    register CHAR *ptr;
    register WORD megabyte;
    WORD ceiling, totalmeminkb;
    DWORD limit;

    /*   Parse the command line. */
    while (++ argv, -- argc) {
         ptr = *argv;
         if (*ptr == '/') {
             if (ptr [1] == '?' AND ptr [2] == '\0') {
                   printf (description);
                   return (1);
             }
             PutError ("invalid switch - %s", ptr);
             return (-1);
         }
         else {
             PutError ("invalid parameter - %s", ptr);
             return (-1);
         }
    }

    /*   All being well, check that the machine is an 80386 (so that the Flat
         Model may be used to access extended memory without relying on an
         interface that automatically enables the A20) and that an XMS server
         has been installed (to control the A20 in the test). */
```

```
if (NOT Is80386 ()) {
      PutError ("requires 80386 CPU or better");
      return (-1);
}
if (InProtectedMode ()) {
      PutError ("CPU must be in real mode");
      return (-1);
}
lpXMS = GetXMSAddress ();
if (lpXMS == NULL) {
      PutError ("no XMS server present");
      return (-1);
}

/*    Compute the number of 1MB boundaries that lie in extended memory
      known to be installed on the machine. Although programs may trap
      int 15h function 88h to change the amount of extended memory
      reported to exist, the original value (disregarding possible
      activity before IO.SYS) is recorded by DOS as a number of K. */

totalmeminkb = GetTotalMemory ();
if (totalmeminkb == 0) {
      PutError ("unable to determine amount of physical memory");
      return (-1);
}
ceiling = (totalmeminkb + 1023) >> 10;
if (ceiling > CAPACITY) ceiling = CAPACITY;
limit = ((DWORD) ceiling << 20) - 1;

/*    Disable interrupts. The particular scheme used for generating
      addresses above 1MB is unstable. Moreover, the test involves
      writing a different value at each megabyte boundary for
      identification, saving whatever was there at the start. The small
      risk that an interrupting process may need this data should not be
      taken. */

ptr = NULL;     /*  The testing routine returns a (non-zero) pointer
            to an error message, to be output once
            interrupts are re-enabled. */

_disable ();
if (GetESLimit () < limit) SetESLimit (limit);

/*    Enable the A20 line to begin the test and restore it afterward.
      Note that the Local Enable function always activates the A20. The
```

```
          matching Local Disable restores the previous state, which does not
          necessarily entail disabling the A20 physically. */

     if (NOT LocalEnableA20 ()) ptr = "cannot enable A20";
     else {
          ptr = TestMemoryMap (ceiling);
          if (NOT LocalDisableA20 ())
               if (ptr == NULL) ptr = "could not restore A20 after test";
     }

     _enable ();

     /*   Display the results. They cannot be presented at the time they are
          gathered because of the need to prevent interrupts. Begin with any
          pending error message. */

     if (ptr) {
          PutError (ptr);
          return (-1);
     }
     printf ("\nMemory mapping caused by disabling the A20 line:\n");
     for (megabyte = 0; megabyte < ceiling; megabyte ++) {
          printf ("\n  %2dMB -> %2dMB", megabyte, disabled [megabyte]);
     }
     printf ("\n");
     return (0);
}

static CHAR *TestMemoryMap (WORD ceiling)
{
     register WORD count, megabyte;

     /*   Write an identification number at each megabyte boundary but first
          save the data that will be overwritten. This function is
          called with the A20 enabled, so the whole address space is
          accessible. */

     for (megabyte = 0; megabyte < ceiling; megabyte ++) {
          originals [megabyte] = ReadID (megabyte);
     }
     for (megabyte = 0; megabyte < ceiling; megabyte ++) {
          WriteID (megabyte, megabyte);
     }

     /*   A single Local Disable function need not actually deactivate the
```

```
        A20 line. This may be done, by calling the Local Disable
        function repeatedly. The number of calls needed should be counted
        so that the local A20 state can be restored later. */

    count = PhysicalDisableA20 ();

    /*  If this has not disabled the A20, try to restore the
        XMS server's local A20 state before returning an error. */

    if (NOT (QueryA20 () == OFF)) {
        if (count) {
            do {
                if (LocalEnableA20 ()) count --;
            } while (count);
        }
        return ("cannot disable A20");
    }

    /*  Now that the A20 line is known to be disabled, retrieve the words
        left at the megabyte boundaries. */

    for (megabyte = 0; megabyte < ceiling; megabyte ++) {
        disabled [megabyte] = ReadID (megabyte);
    }

    /*  Enable the A20 again (restoring the local state) and replace the
        words that were originally at the megabyte boundaries. */

    do {
        if (LocalEnableA20 ()) count --;
    } while (count);

    if (QueryA20 () == OFF) return ("cannot reenable A20 to restore data");

    for (megabyte = 0; megabyte < ceiling; megabyte ++) {
        WriteID (megabyte, originals [megabyte]);
    }

    return (NULL);
}

static WORD PhysicalDisableA20 (void)
{
    register WORD count;
    /*   To disable the A20 using XMS functions, call the Local Disable A20
         service repeatedly. This should eventually disable the true A20
         line, causing the next Local Disable to fail.

         Counting the Local Disable calls yields the XMS server's internal
         A20 counter, which is returned as this C function's value. Each
         Local Disable must be matched by a Local Enable when restoring the
         A20 to its original state.

         Since the Local Disable call may fail because of an A20 hardware
         problem, the A20 status should be tested after this function
         returns, to confirm that the A20 has in fact been disabled. */

    if (QueryA20 () == OFF) return (0);

    /*   Limit the number of times that LocalDisableA20 () is called, just
         in case it never fails. */

    count = 0x0100;
    while (LocalDisableA20 () AND -- count) {
    }
    return (0x0100 - count);
}

static WORD GetTotalMemory (void)
{
    if (_osmajor < 4) return (0);

    _asm {
            mov     ah,52h
            int     21h
            mov     ax,es:[bx].wExtendedMemoryInK
            add     ax,1024
            jnc     done
            xor     ax,ax
        done:
    }
}

static void PutError (CHAR *str, ...)
{
    va_list va_start (argptr, str);
    printf ("\nA20MAP error:  ");
    vprintf (str, argptr);
    printf ("\n");
    va_end (argptr);
}
```

Simple routines for reading and writing a word at an arbitrary megabyte boundary require 32-bit addressing, which is not supported by the Microsoft C compiler (version 6.0). Hence you need the following short routines in assembler. The caller must ensure that the segment limit currently in force for ES will be large enough to cover the megabyte boundary from linear address zero.

```
; RDWRID.ASM

.MODEL                        SMALL, C
.386

;   ====================================================================

.CODE

ReadID                        PROC     PUBLIC megabyte:WORD

                              xor      ax,ax
                              mov      es,ax
                              mov      bx,[megabyte]
                              shl      ebx,20
                              mov      ax,es:[ebx]
                              ret

ReadID                        ENDP

;   --------------------------------------------------------------------

WriteID          PROC         PUBLIC megabyte:WORD, value:WORD

                              xor      ax,ax
                              mov      es,ax
                              mov      bx,[megabyte]
                              shl      ebx,20
                              mov      ax,[value]
                              xchg     ax,es:[ebx]
                              ret

WriteID          ENDP

END
```

The A20MAP program requires the GETXMS module needed by practically all programs in Part 3 and the GETLIMIT, IS80386, SETLIMIT, and VM modules from Chapter 11, in addition to the main A20MAP.C file and the small RDWR.ASM file.

A20MAP determines the extended memory map when the A20 line is disabled. The present implementation has a granularity of one megabyte and requires an 80386 processor or better, operating in real mode. On a machine that controls the A20 by passing the address line through an AND gate with a signal from some bit at an I/O port, the A20MAP program should produce a map similar to

```
Memory mapping with disabled A20 line:

    0MB  ->   0MB
    1MB  ->   0MB
    2MB  ->   2MB
    3MB  ->   2MB
    4MB  ->   4MB
    5MB  ->   4MB
    6MB  ->   6MB
    7MB  ->   6MB
```

showing wrap-around for every second megabyte. It is also possible to include other address lines in the controlling mechanism, which may reduce the incidence of wrap-around, as with a Compaq Deskpro

```
    0MB  ->   0MB
    1MB  ->   0MB
    2MB  ->   2MB
    3MB  ->   3MB
    4MB  ->   4MB
```

or, conceivably, to collapse the entire address space to the first megabyte:

```
    1MB  ->   0MB
    2MB  ->   0MB
    3MB  ->   0MB
```

Only in the second case is it possible that A20 manipulation affects no memory outside the HMA (the program would have to be written with finer resolution than 1MB in order to determine this for certain). Knowledge of the way that disabling the A20 affects memory addressing on a particular machine may be very important if, for example, an unchecked call to disable the A20 has returned an error but an extended memory disk cache has continued operation. This is no problem to a computer on which the A20 state has no consequences for memory above the HMA, but for machines in the first and third cases, a single unchecked A20 failure could invalidate megabytes of memory at a stroke.

HIMEM's A20 Handlers

Hardware dependency is met by endowing HIMEM.SYS with a growing number of embedded A20 handlers, one of which is selected during the device driver's initialization and given sole responsibility for controlling the A20. Of course, all unused handlers are discarded from memory.

There is one circumstance in which HIMEM will install itself but play absolutely no role in A20 management. The **/a20control:off** switch directs HIMEM to take control only if the A20 is off. If the A20 is found to be enabled already, suggesting either that the user would like the A20 to be permanently enabled or that some program loaded earlier controls the line (presumably through another interface), then HIMEM displays an appropriate message and proceeds with its initialization, but thereafter its XMS functions for A20 control indicate success trivially.

From the user's perspective, an A20 handler may be specified by supplying a name or equivalent number through HIMEM's **/machine** switch. The catalog of available handlers presented as Table 9-1 reads like a mini-history of Microsoft's industrial co-operation. Separate A20 routines for the standard AT, the PS/2, and the AT&T 6300 Plus date back to a time when RAMDRIVE arranged its own extended memory access and the early releases of HIMEM.SYS used int 2Fh function 16h rather than 43h. The AT&T machine belongs to the same family as the Olivetti computers that have (for many years) received particular attention in the graphics library of Microsoft's C compiler. Support for the (not quite PC-compatible) HP Vectra—added not long after the others—seems a natural consequence of Hewlett-Packard's work with Windows, more direct indications of which may be seen in EGA.SYS and some screen grabbers (developed for Windows but now also distributed with DOS 5), and in many Windows modules such as SYSTEM.DRV.

Some of the embedded handlers are accompanied by (non-resident) routines that try to establish their compatibility with the computer in question, perhaps by checking that some meaningful letters are at an expected location in the ROM BIOS. These identification routines also perform any necessary initialization (for instance, to disable the A20 line on the PS/2). If the machine type is specified explicitly, HIMEM calls the corresponding handler's initialization routine but disregards the result of the identification test. Otherwise (but with exception the following), it attempts an automated identification on the user's behalf. As noted in Table 9-1, only some handlers are able to establish their own suitability; they are consulted in the following order:

ptlcascade, att6300plus, ps2, hpvectra, toshiba, zenith, philips, css, fasthp, bullmicral (version 3.07), dell, at

Table 9-1: A20 Handlers Supported by HIMEM.SYS Versions 2.77 and 3.07 (Released with DOS 5 and Windows 3.1, Respectively)

NAME	NUMBER	SUBNUMBER	SELF-IDENTIFYING	FIRST KNOWN RETAIL HIMEM VERSION
at	1	0, 1 or 2	yes	
ps2	2		yes	
ptlcascade	3		yes	
hpvectra	4		yes	
att6300plus	5		yes	
acer1100	6		no	2.50
toshiba	7	2	yes	2.60
wyse	8		no	2.60
tulip	9		no	2.77
zenith	10		yes	2.77
at1	11	0	no	2.77
at2	12	1	no	2.77
css	12	1	yes	2.77
at3	13	2	no	2.77
philips	13	2	yes	2.77
fasthp	14		yes	2.77
ibm7552	15		no	2.77
bullmicral	16		yes, in 3.07	2.77
dell	17		yes	3.07

The AT handler—the last to be consulted and, in that sense, the default machine type—is configurable. There is a family of AT handlers, marked by the subnumbers in Table 9-1, which are also reported during HIMEM's installation, such as

```
Installed A20 handler number 13,2
```

for AT3 or PHILIPS. All three variants control the A20 through bit 1 of the 8042 keyboard controller's discrete data port 60h (yes, you read that right: A20 is usually manipulated via the *keyboard* controller) but differ in their behavior afterward. The AT2 handler is the fastest—the others either send another command to the controller or wait for an indication that the data has been accepted. In some cases, the delay may be necessary for reliable performance. A specific choice entered on the command line is adopted without question, but if the preference is expressed vaguely with the **/m:at** switch or if HIMEM is left to decide for itself, then each variant is tried in turn—1024 cycles of enable, test, disable, test.

Some difficulty may strike users of computers on which no AT variant works perfectly. In the absence of a **/machine** switch and with no other embedded handler having claimed responsibility for A20 management, HIMEM reports its inability to control the A20 line and refuses to install. If, the **/m:at** switch is used (for instance, without explicit selection of a subhandler), then difficulty may turn to confusion as HIMEM disregards the unsatisfactory performance result and installs the last AT variant that it tested—the AT3 hander, this time identified by the report

```
Installed A20 handler number 1,3
```

The reason why specific designation of an AT handler may convince a recalcitrant HIMEM to install is that in each call to a Local A20 function, physical manipulation of the A20 may be attempted up to three times, so the 100% performance demanded by the default test for AT installation is perhaps more stringent than it need be. The machine may seem to work perfectly well, but HIMEM would have rejected it without the explicit **/m:at** switch.

As a further instrument of configuration, all members of the AT family (and only these handlers) are subject to the **/cpuclock:on** switch, which seeks to preserve other bits at the keyboard controller's discrete data port when toggling the bit that masks the A20. On many computers, these other bits are unused and preservation is not needed.

At the programming level, each A20 handler is a procedure that offers two or possibly three functions for A20 control, as shown in Table 9-2.

Table 9-2: HIMEM's Calling Convention for A20 Handlers

DISABLE/ENABLE A20

INPUT:

AX	0000h	to disable
	0001h	to enable

SUCCESS:

ax	0001h

Table 9-2: Continued

FAILURE:

ax	0000h

Query A20 Status

INPUT:

AX	0002h

OUTPUT:

ax	0000h	if A20 is disabled
	0001h	if A20 is enabled

All A20 handlers must be able to disable and enable the address line, but the ability to determine the A20 status is not mandatory. Of the predefined handlers, only those for the ps2, ptlcascade, and ibm7552 support the query function.

In HIMEM version 2.77 and earlier, the choice of handler was reported during device-driver initialization only, but version 3.07 introduces an extension to the int 2Fh portion of the XMS interface so that software may ask HIMEM about the installed A20 handler as shown in Table 9-3.

Table 9-3: HIMEM's Int 2Fh Support for Identifying A20 Handlers

INPUT:

AX	4308h

OUTPUT:

al	43h
bh	A20 handler number from Table 9-1; or 00h for an external handler (see below)
bl	A20 handler subnumber from Table 9-1, defaulting to 00h

External A20 Handlers

For some machines no handler supplied within the HIMEM.SYS file is able to control the A20 line reliably. To accommodate this possibility, HIMEM versions 2.77 and later check for the provision of an external A20 handler. This is done via yet another (apparently undocumented) int 2Fh function.

If no machine type is specified on the command line, HIMEM issues this int 2Fh request for an external handler before trying to identify the machine itself. In practical terms, the facility requires a program that can be loaded as a device driver before HIMEM and that retains both a custom A20 handler and an int 2Fh routine through which HIMEM will discover the A20 handler waiting for adoption. The external handler must subscribe to the same interface as the handlers embedded in HIMEM but will be called as a far procedure. In a sense, a favorable response to this int 2Fh call creates a user-defined machine type and offers the next best thing to having in-built support. Cynics may observe that extensibility of this sort is somewhat pointless if its existence is kept secret (source code for HIMEM.SYS is freely available, but the author's request was met with version 2.06—dating from long before the introduction of the external A20 facility).

Table 9-4: HIMEM's Int 2Fh Request for an External A20 Handler

INPUT:

AX	4330h

OUTPUT:

al	80h	
es:bx	address of external A20 handler	
cl	00h	if handler cannot report A20 status
	01h	if handler supports the function to Query A20 Status (any non-zero value for cl suffices, but HIMEM's internal handlers use the value 01h)

Faced with a machine whose method of A20 control is not known to HIMEM—a support problem that has far from vanished—one solution, pending a begging letter to the manufacturer and/or Microsoft, is to write a custom A20 handler and build it into a small program to be loaded as a device driver before HIMEM. The necessary int 2Fh routine is easily written as a resident module of a program XA20.EXE that expects a machine-specific A20 handler to be supplied in another module at link-time. XA20.EXE loads the external A20 handler for subsequent installation by HIMEM.SYS.

```
/* XA20.C */

#include  <intpack.h>
#include  <keepc.h>
#include  <standard.h>

extern BOOL _fastcall _far ExternalA20Handler (int);

BYTE A20DetectionFlag = FALSE;
void (_interrupt _far *lpOrgInt2Fh) (void) = 0;

void _interrupt _far int2Fh (struct INTPACK r)
```

```
{
  if (r.ax == 0x4330) {
          r.es = (_segment) ExternalA20Handler;
          r.bx = (unsigned) ExternalA20Handler;
          r.cl = A20DetectionFlag;
          r.al = 0x80;
  }
  else ChainIntr (lpOrgInt2Fh);
}
```

Some non-resident code is needed in order to load the int 2Fh routine and the external A20 handler into memory (although it is faintly possible that the latter can be avoided—if the A20 handler in ROM, used by the BIOS code for int 15h functions 87h and 89h, has exactly the calling protocol required by HIMEM and therefore needs no support beyond the int 2Fh function that tells HIMEM of the A20 handler's existence). All command-line arguments are passed to a machine-specific function charged with establishing the suitability of installation and configuring the external handler.

```
/* main.c - initialization for XA20.EXE */

#include  <dos.h>
#include  <stdarg.h>
#include  <stdio.h>

#include  <keepc.h>
#include  <standard.h>

static CHAR description [] = "\n\
Loads an external A20 handler for subsequent installation by HIMEM.SYS\n\
";

/* A nonresident routine to test the hardware for suitability and the
  address of a string to describe the configuration must be
  provided, specially for the machine concerned, but may be NULL (as will
  be the case if they are not linked) if default behavior is acceptable. */
extern BOOL InitA20Handler (int, char **);
extern CHAR *pMachineName;

/* The A20 handler in a resident segment and an associated flag must
  be provided especially for the machine concerned. */
extern BOOL _fastcall _far ExternalA20Handler (int);

/* External functions and data concerned with communicating the existence
  of the external A20 handler. */
extern void (_interrupt _far * _RESIDENT lpOrgInt2Fh) (void);
extern void _interrupt _far int2Fh (void);

/* A function to determine that an XMS server is not yet present */
extern void (_far *GetXMSAddress (void)) (void);

/* A function to check that the computer supports an A20 line */
static BOOL IsAtLeast80286 (void);

/* A Boolean flag to distinguish normal program execution from device
  driver initialization */
extern BOOL _RESIDENT bLoadedAsDriver;

/* Message output */
void PutError (CHAR *, ...);
void PutWarning (CHAR *, ...);

/* ========================================================================== */

int main (int argc, char **argv)
{
  /*      Check that the program has been linked with an A20 handler. */
  if ((void _far *) ExternalA20Handler == NULL) {
          printf (description);
          printf ("\nThis program must be linked with machine-specific code\n");
          return (-1);
  }

  /*      Set a default machine description. */
  if (pMachineName == NULL) pMachineName = "some un-named machine";

  /*      If an XMS server is already active, it is too late to load an
          A20 handler for HIMEM to install. Regard this condition as a fatal
          error if the program is being loaded as a device driver;
          otherwise describe the program, taking care to indicate its
          machine-dependent nature. */

  if (GetXMSAddress ()) {
          if (bLoadedAsDriver) {
            PutError ("XMS handler already installed");
            return (-1);
          }
          else {
            printf (description);
            printf ("\nThis program has been prepared specifically for %s\n",
                    pMachineName);
```

```
                return (0);
            }
    }

    /*      Do not proceed if installation would be clearly pointless. */
    if (NOT IsAtLeast80286 ()) {
            PutError ("requires 80286 processor or better");
            return (-1);
    }

    /*      If a function for identifying the hardware or checking suitability
            has been linked, call it. Displaying an error message is the
            responsibility of that function. */

    if ((void *) InitA20Handler != NULL AND NOT InitA20Handler (argc, argv))
            return (-1);

    /*      Install the int 2Fh service through which HIMEM will discover the
            external A20 handler, then declare success and tidy up. */

    lpOrgInt2Fh = _dos_getvect (0x2F);
    _dos_setvect (0x2F, int2Fh);

    printf ("\nExternal A20 handler configured for %s\n", pMachineName);

    CloseAllFiles ();
    FreeEnvironment ();
    ExitKeepC (0, 0);
}

static BOOL IsAtLeast80286 (void)
{
    _asm {
            pushf                   ; Preserve the FLAGS register so
                                    ; that IF can be restored at the end.

            xor     ax,ax           ; Try clearing all FLAGS bits.
            push    ax
            popf

            pushf                   ; On the 8086 and 80186, bits C to F
            pop     ax              ; are permanently set.
            and     ax,0F000h
            cmp     ax,0F000h
            jz      old
            mov    ax,0001h
            jmp    done
          old:
            xor    ax,ax
          done:
            popf
    }
  }

void PutError (CHAR *str, ...)
{
  va_list va_start (argptr, str);
  printf ("\nExternal A20 fatal error:");
  vprintf (str, argptr);
  printf ("\n");
  va_end (argptr);
}

void PutWarning (CHAR *str, ...)
{
  va_list va_start (argptr, str);
  printf ("\nExternal A20 warning:");
  vprintf (str, argptr);
  printf ("\n");
  va_end (argptr);
}
```

The remaining modules in the XA20 program must be prepared for the machine at hand. The ROM BIOS (unless it has been fitted to an incompatible computer) will certainly contain routines for controlling the A20 line. An easy way to find them, pending advice from the computer's manufacturer, is to examine the instructions used for int 15h functions 87h or 89h—the BIOS services that move data between locations anywhere in memory or that create a minimal protected mode environment—since both of these must either contain or call code to handle the A20 line.

The following module, HANDLER.C, enables and disables A20 using a technique close to that used by the HIMEM.SYS AT2 method: A20 is manipulated via the 8042 keyboard controller ports 60h and 64h. Port 64h is the 8042 control port (when written) and status port (when read); Port 60h is the data port. Commands are sent to port 64h to tell the 8042 what to do with data read from or written to port 60h. The commands for accessing the keyboard controller discrete data are D0h to read and D1h to write. For AT machines, bit 1 of the discrete data corresponds to the A20 gate.

Thus, HIMEM can disable the A20 by first writing D1h to port 64h (to tell the 8042 that port 60h is about to be used for discrete data) and then writing DDh (bit 1 clear) to port 60h. The variants of AT handler differ by what they do after this: AT1 writes FFh to port 64h as if to reset the 8042 command mode, and AT2 does nothing (making it identical to the WYSE/TULIP handler that predates it).

A20 is controlled in other ways on non-AT machines, of course, (this being the whole point of having different A20 handlers. For example, on an HP Vectra you can control the A20 just by writing the DDh/DFh values to port 64 twice in succession. Weird!

```
/* handler.c */

/* This file contains a sample external A20 handler, close in form to the
  handler HIMEM.SYS provides as AT2. */

#include  <conio.h>
#include  <dos.h>

#include  <keepc.h>
#include  <standard.h>

#define   KEYBRD_DATA      0x60
#define   KEYBRD_STATUS    0x64     /* in port 0x64 */
#define   KEYBRD_CMD       0x64     /* out port 0x64 */

/* commands to port 0x64 */
#define   READ_OUT_PORT    0xD0
#define   WRITE_OUT_PORT   0xD1

#define   DISABLE_A20      0xDD
#define   ENABLE_A20       0xDF

/* An A20 handler may support up to three functions. */

#define   DISABLE  0
#define   ENABLE   1
#define   QUERY    2

/* HIMEM calls the last function (QUERY) only if the ability to service it
  was indicated when the A20 handler's existence was reported via int 2Fh,
  i.e., if the A20DetectionFlag was set to TRUE by InitA20Handler ().

  Although HIMEM's A20 handler for AT machines does not support a query
  facility, keyboard controllers may be able to report the A20 status as
  well as change it. Compile with the symbol _QUERY_OK_ to include code
  for this purpose. */

/* The following pragma makes sure the compiler will generate INP and OUTP
```

```
  instructions in-line. */
#pragma   intrinsic (inp, outp, _disable)

/* The keyboard controller sets bit 1 at port 64h while it is busy
  processing data written to it. The following macro generates in-line
  code to wait for the keyboard controller to clear that bit. */

#define wait_while_busy () \
  for (count = 0; inp (KEYBRD_STATUS) & 2 AND -- count; ) {}

/* Likewise, when data is available from the keyboard controller, it sets
  bit 0. */

#define wait_for_data () \
  for (count = 0; NOT (inp (KEYBRD_STATUS) & 1) AND -- count; ) {}

BOOL _fastcall _far ExternalA20Handler (int code)
{
  register WORD count;
  int control;

  #ifdef _QUERY_OK_
  if (code == QUERY) {
         _disable ();
         wait_while_busy ();
         outp (KEYBRD_CMD, READ_OUT_PORT);
         wait_while_busy ();
         wait_for_data ();
         control = inp (KEYBRD_DATA);
         return (control & 2 ? TRUE : FALSE); // Bit 1 is A20 mask
  }
  #endif

  control = (code == DISABLE) ? DISABLE_A20 : ENABLE_A20;
  _disable ();
  wait_while_busy ();
  outp (KEYBRD_CMD, WRITE_OUT_PORT);
  wait_while_busy ();
  outp (KEYBRD_DATA, control);
  wait_while_busy ();
  return (count ? TRUE : FALSE);
}
```

For this simple demonstration, no initialization is needed nor is verification of the machine's identity attempted.

```
/* machine.c */

/* This file contains a (minimal) sample InitA20Handler function, called
  during the XA20 device driver's initialization. */

#include  <keepc.h>
#include  <standard.h>

/* The external A20 handler and an associated flag are defined in the
  HANDLER.C source file. */

extern BOOL _fastcall _far ExternalA20Handler (int);
extern BYTE _RESIDENT A20DetectionFlag;

#define   DISABLE  0
#define   ENABLE   1
#define   QUERY    2

/* The machine for which the external A20 handler is prepared may be
  identified to the user in one of two circumstances. If the XA20 device-
  driver file is executed as a normal DOS program (resulting in a program
  description) or on successful initialization when actually loaded as a
  device driver. Default messages will be used in both cases if the
  following variable is a NULL pointer, but in the interests of clarity,
  it is recommended that a description be provided instead. */

CHAR *pMachineName = "Geoff's cheap 386 clone";

/* The function called to initialise the external A20 handler must

  1. set the A20DetectionFlag to TRUE if the external A20 handler
     supports the Query A20 Status function

  2. physically disable the A20 line if it is already enabled

  in addition to possibly establishing the suitability of the machine on
  which the program is being executed (and/or testing that the proposed
  A20 handler does in fact work).

  Command-line arguments are passed to InitA20Handler for general
  configurability (not taken up in this example). Standardized error
  messages may be displayed using the PutError or PutWarning functions,
  which follow the style of printf (). */

extern void PutError (CHAR *, ...);
extern void PutWarning (CHAR *, ...);

BOOL InitA20Handler (int argc, char **argv)
{
  #ifdef _QUERY_OK_
  A20DetectionFlag = TRUE;
  #endif

  ExternalA20Handler (DISABLE);
  return (TRUE);
}
```

As with other projects using this book's scheme for resident programming, a makefile is more or less essential:

```
!INCLUDE            <standard.mak>
!INCLUDE            <keepc.mak>

TARGETS : xa20.exe

#  The first two modules are machine-dependent, i.e., replacements must be
#  supplied if developing the XA20 program for A20 hardware other than the
#  standard catered for in the samples. The symbol QUERY_OK on the NMAKE
#  command line configures the sample A20 handler to support A20 queries.

!IFDEF QUERY_OK
_QUERY_OK_ = /D _QUERY_OK_
!ENDIF

handler.obj : handler.c
 $(COMPILE) $(KEEP) $(_QUERY_OK_) handler.c

machine.obj : machine.c
 $(COMPILE) $(_QUERY_OK_) machine.c

#  The next two modules provide the means to load the machine-dependent
#  A20 handler into memory ready for adoption by HIMEM.SYS. These need
#  no alteration when developing the XA20 program for different A20
#  hardware.

main.obj : main.c

xa20.obj : xa20.c
 $(COMPILE) $(KEEP) xa20.c
```

```
#  Link with DRIVER.OBJ, CRTDRVR.LIB and CRTKEEPC.LIB

xa20.exe : handler.obj machine.obj main.obj xa20.obj
 $(LINK) driver+crtdrvr.lib+crtkeepc.lib+ \
          getxms+handler+machine+main+xa20,xa20.exe;
```

The finished driver must be loaded before HIMEM.SYS and needs roughly a kilobyte of low memory, depending on the amount of code and data that accompanies the A20 handler—a small price to pay if presented with a machine whose A20 hardware is not yet supported by HIMEM.

CHAPTER 10

Extended Memory Sources

For historical reasons, the XMS cannot be integrated into DOS as an essential system component that would be loaded automatically on machines fitted with extended memory. By the time the XMS had been proposed to developers, there had already appeared a significant amount of software that obtains and uses extended memory through other methods. Although such software is surely diminishing in quantity, its possible execution must be a defensive consideration in the design of any XMS server.

Contrast this position with that of conventional memory. When DOS initializes its memory management facility, it relies on the ROM BIOS to have determined the amount of RAM fitted as a single block beginning at address zero and to supply the answer (in kilobytes) through an int 12h routine. This routine and the word at 0040:0013 in which a PC-compatible BIOS keeps the return value remain available to DOS programs on the understanding that physical availability of RAM is not a licence for use—a management scheme for conventional memory is already in place by the time any of DOS's clients are executed.

Int 15h

The BIOS also assesses how much extended memory has been installed (as a single block beginning at 1MB), reporting this result via int 15h function 88h (Get Extended Memory Size). However, with no one system component having clear responsibility for extended memory management, programs needing to use extended memory have little choice but to change the value returned by the int 15h function, claiming the difference for themselves and letting subsequent programs think that less extended memory has been fitted to the computer. The only direct role played by DOS is to record the int 15h assessment of installed extended memory before any user-specified code has the opportunity to change it.

This is not unlike the way that a typical boot sector virus (or the BIOS itself, if supporting an Extended BIOS Data Area) adjusts the word at 0040:0013 and hence the behavior of int 12h to protect a small area at the top of conventional memory from absorption into the DOS management scheme. In the case of extended memory, the BIOS is not constrained to keep the int 15h return value at some fixed location for backward compatibility, so int 15h function 88h (Get Extended Memory Size) can be made to return a different result only by hooking the interrupt—as demonstrated by the resident programming sample (INT15MEM) in Chapter 6 and its extension to a device driver in Chapter 7.

The obligation on XMS servers when claiming the pool of extended memory that they will subsequently disburse among their clients is to trap int 15h function 88h to hide the pool from non-XMS programs that may be executed later. Even if the XMS server is intended for a market in which sales prospects would not be damaged by the refusal of older, non-XMS software to initialize (thinking there to be no extended memory), the value returned by int 15h function 88h must still be altered for safety.

VDISK

As a method for securing extended memory, the int 15h alteration at least has the advantage of minimalism: It relies on nothing other than the machinery already used by the BIOS to communicate the presence of extended memory. Sadly, XMS servers must guard themselves against another method that seems to have been promoted by IBM's distribution of a source listing for their VDISK driver as part of the normal PC-DOS retail package. The VDISK driver uses int 15h function 88h to discover how much extended memory has been installed, but instead of trapping int 15h to change the value seen by programs loaded afterward, it expects other programs to examine special locations for a description of VDISK's own usage.

The first extended-memory VDISK creates its "disk" at the 1MB boundary, to be followed by disk buffers for any subsequent VDISK installations. The VDISK source listing that accompanied PC-DOS 3.20 suggests that other users of extended memory find the end of all VDISK buffers by consulting a field immediately after the BIOS Parameter Block (BPB) in the boot sector of this first VDISK, having first established its validity by matching the "VDISK" characters in the boot sector's OEM field. The debugger dump

```
FFFF:0010 00 00 00 56 44 49 53 4B-33 2E 33 80 00 01 01 00 ...VDISK3.3.....
FFFF:0020 01 40 00 00 02 FE 06 00-08 00 01 00 00 00 40 04 .@...~........@.
```

shows a typical VDISK boot sector at the 1MB boundary, with the identification string at offset 03h. The reservation of 64KB is indicated by the word at offset 1Eh, which is the address (measured in K) of the first extended memory left unused.

Curiously, although successive copies of VDISK update this field to reserve additional extended memory, they do not examine it when seeking the address of the extended memory that has not yet been used by earlier VDISKs. Instead, they try to locate the low memory portion of the first VDISK driver, where immediately after the device driver header there should be a fragment of a volume label ready for entry in the root directory of the "disk", and after that (at offset 2Ch) there should be the 24-bit linear address of the lowest available extended memory:

```
1183:0000 00 00 CF 0F 00 08 A9 00-D4 00 01 00 00 00 00 00 ..O...).T.......
1183:0010 00 00 56 44 49 53 4B 20-20 56 33 2E 33 28 00 00 ..VDISK V3.3(..
1183:0020 00 00 00 00 00 00 00 00-00 60 86 09 00 00 11 00 .........`......
```

The first installed VDISK driver is expected to hook int 19h, then subsequent instances attempt to locate the first by inspecting the segment pointed to by that vector. If that segment contains the expected VDISK volume label, then the 24-bit address at offset 2Ch is taken as the site for the new "disk" in extended memory, the field is updated and then converted into kilobytes for entry into the first VDISK's boot sector at the fixed address 10001Eh.

The integrity of this scheme is wholly dependent on successive VDISK drivers being able to locate the first one via int 19h. The problem with this is that, as discussed in Chapter 2, all programs that hook interrupt vectors have obligations to also hook int 19h (though thankfully, these will often be covered by software loaded earlier). These obligations arise as intrinsic consequences of the design of the PC's BIOS, which permits the operating system to be reloaded without a BIOS re-initialization. These obligations must be faced even by programs that have nothing to do with extended memory. It is hard to understand why such programs should have to accommodate a folly made by some other program entirely irrelevant to their operations, but make no mistake, VDISK is essentially requiring that if a program hooks int 19h, then it must also place a fake VDISK header at the start of the segment containing the int 19h handler. Should some intervening program hook int 19h without meeting this unreasonable imposition, the next VDISK driver to be installed will believe itself to be the first and will set up its extended memory "disk" at 1MB, destroying all trace of any extended memory allocation recorded by earlier VDISKs.

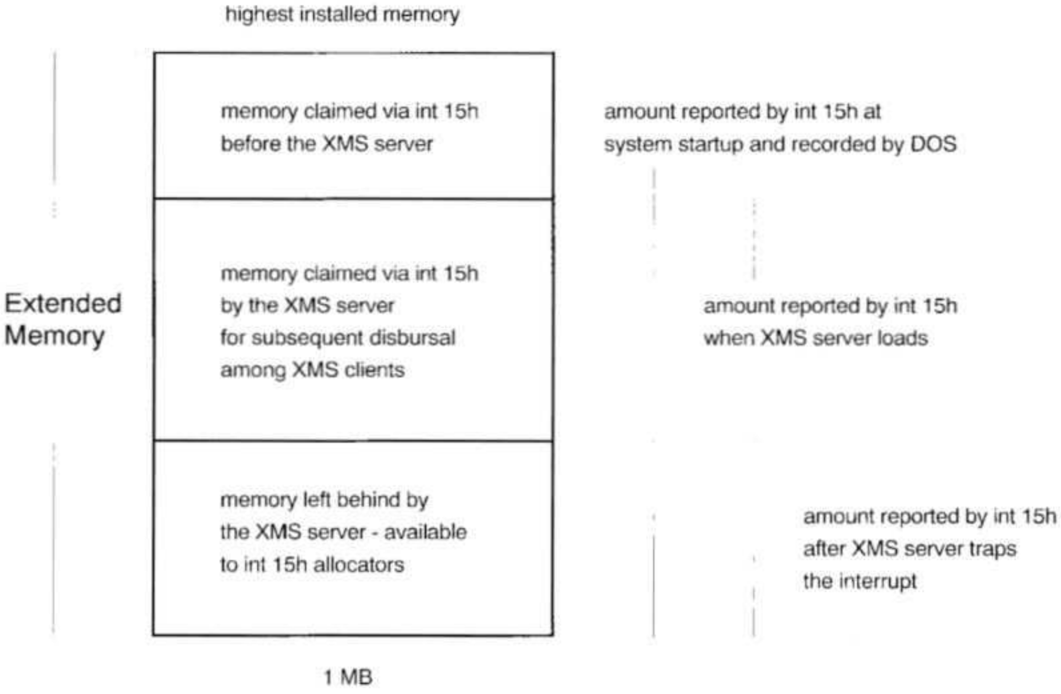
The bottom line is that multiple loading of VDISK—at least in the form prepared for PC-DOS 3.20 and 3.30—is unsound, so that even if a programmer seeking extended memory were inclined to mimic the VDISK allocation method for compatibility, it would be unsafe to do so. Admittedly, multiple loading is an unlikely occurrence and programs that imitate VDISK probably act more defensively than the real thing, but with no reliable way to determine how much extended memory has been claimed by VDISK allocators, the presence of the characters "VDISK" at the address FFFF:0013 or at offset 12h in the segment pointed to by int 19h should be regarded with suspicion by any extended memory user and as fatal to the installation of a bullet-proof XMS server; compatibility is not an issue.

The wonder is not that someone can have designed VDISK with this flaw, nor even that a major player in the PC market promoted it as a model for extended memory allocation. After all, DOS may have seemed to be expiring at the time (late 1985) and was not considered a sufficiently important project to justify the little reflection needed for a proper job. But how is it that writers with good reputations and VDISK source code freely available for inspection (recall that PC-DOS used to come with VDISK.LST) can have played along more or less unquestioningly? For example, as late as 1990, the authors of *Extending DOS*, while not singing VDISK's praises, still suggested that programmers might follow VDISK's method of extended memory allocation and came nowhere near to hinting any danger.

Cooperation

It is an easier matter for an XMS server to coexist with int 15h allocators. Since an XMS server *must* trap int 15h to secure its own pool of extended memory, it costs little more to provide an option through which some designated amount of extended memory may be left out of the XMS pool (see Figure 10-1).

Figure 10-1: Extended Memory Allocation by Hooking Int 15h Function 88h


highest installed memory
memory claimed via int 15h before the XMS server
amount reported by int 15h at system startup and recorded by DOS
Extended Memory
memory claimed via int 15h by the XMS server for subsequent disbursal among XMS clients
amount reported by int 15h when XMS server loads
memory left behind by the XMS server - available to int 15h allocators
amount reported by int 15h after XMS server traps the interrupt
1 MB

In the case of HIMEM (the standard XMS implementation and therefore one that must pay particular attention to compatibility with older software), this option may be exercised through the **/int15** switch, together with the number of kilobytes to be left behind for other int 15h allocators. Some XMS servers—including (older) OEM variants of HIMEM—provide the same facility under another name.

It only with the release of DOS 5 that HIMEM was offered this simple form of coexistence, previously preferring to delay hooking int 15h until some program actually declared itself to be an XMS client by making a nontrivial XMS request. This method—advocated in the implementation notes that accompany the XMS documentation but not adopted by all XMS servers—has the advantage of accommodating other int 15h allocators without having to know their memory requirements in advance. The effect is not much different from loading these other programs before HIMEM and is of no benefit at all to an int 15h allocator that cannot be run before the first XMS client.

For evidence of this delay, reboot with a CONFIG.SYS file in which the INT15MEM device driver sample from Chapter 7 is loaded immediately after HIMEM:

```
device=himem.sys
device=int15mem.exe 16
dos=low
```

Even though HIMEM is not configured to reserve memory for int 15h allocators, the INT15MEM driver still receives its token 16KB of extended memory, demonstrating that HIMEM does not claim the computer's contiguous extended memory for the XMS pool straight away. If, however, some XMS client intervenes, as with

```
device=himem.sys
device=smartdrv.sys 128
device=int15mem.exe 16
dos=low
```

then INT15MEM will discover that no extended memory contiguous to the 1MB boundary is available anymore: HIMEM will already have claimed the extended memory in response to XMS requests from its first client (SMARTDRV.SYS in this case).

The **dos=low** statement is important in these examples. Under DOS 5 with **dos=high**, the first XMS client may well be DOS itself, since the SYSINIT module tries to obtain the high-memory area for the DOS kernel code as soon as possible after (if not during) the initialization of an XMS server. Thus, with the CONFIG.SYS file

```
device=himem.sys
device=int15mem.exe 16
dos=high
```

there is effectively no delay in the XMS server's seizure of extended memory, with the result that neither INT15MEM nor any other int 15h allocator is able to obtain extended memory unless it can be loaded before HIMEM. With the release of DOS 5, it became imperative that HIMEM support the explicit reservation of extended memory for int 15h allocators and not simply delay its own int 15h hook.

HIMEM versions distributed with DOS 5 and Windows 3.1 support both techniques (the **/int15** switch and the delayed hook), but a possible conflict between the methods seems to have escaped Microsoft's attention, despite revision of the relevant coding. Clearly, the amount of memory left aside for the int 15h interface cannot exceed the amount available at the initialization of the XMS server. HIMEM rejects any **/int15** value that is too large, but in the context of delaying the int 15h hook, the real comparison is between the amount to reserve and the amount still available via int 15h when the XMS server actually claims the XMS pool. Unfortunately, HIMEM neglects this second check and may compute a negative size for the XMS pool when the time finally comes to hook int 15h. The consequences may be demonstrated with reasonable safety using the following CONFIG.SYS fragment, in which HIMEM.SYS is to leave some extended memory that the INT15MEM program from Chapter 7 is to take:

```
comment //
device=himem.sys /i=64      // if 64 <= leave <= capacity
device=int15mem.exe 7105     // and 0 <= take < capacity,
device=ramdrive.sys 32767 /e  // then problem if capacity < take + leave
dos=low
```

The particular values (leave = 64 and take = 7105) represent the extreme case for a machine with 7MB of extended memory, but the problem domain is described more generally in the comments. INT15MEM eats all but 63KB of extended memory before HIMEM hooks int 15h. Since HIMEM is to leave behind 64KB for the int 15h interface, it "deduces" that the XMS pool must be 63KB minus 64KB = 65,535KB! This results in the successful allocation by RAMDRIVE of 32MB (the maximum disk size supported by the DOS 5 version) even though far less memory is actually present. Clearly, this is not a stable configuration.

Noncontiguous Extended Memory

The value returned by int 15h function 88h should describe a singly-connected region beginning at 1MB, but on some computers memory above 1MB might be fitted in disjoint pieces. Several factors may complicate or even prevent an XMS server's incorporation of noncontiguous extended memory into the XMS pool. There is, a high degree of hardware-dependency involved in knowing that a particular computer supports extra memory. The general difficulty is that this noncontiguous memory does not exist without reason: The XMS server is the interloper and must face any prior claims on the memory.

In the most demanding cases the XMS server is not the only software that seeks to use the non-contiguous memory. In these situations some means must be found both to establish that the memory is not already being used by competing software and to protect the memory from conflict with nonclients after its addition to the XMS pool. An example is provided by certain Compaq machines that have memory installed in the 16th megabyte (sometimes called Built-In Memory, or BIM) as a form of built-in EMS support. Here the amount of memory and its location are described exclusively through a data structure (held in EPROM, as it happens) that may be altered to simulate the memory's nonexistence, effectively gaining it for XMS usage by deluding the EMS routines that the memory has not been fitted.

It probably surprises nobody to hear that HIMEM.SYS knows of this Compaq peculiarity. Version 3.09 of HIMEM.SYS, distributed with DOS 6, has knowledge of another such Compaq peculiarity: Compaq machines without an EISA signature may reveal via int 15h function E801h that they support number of 64KB memory blocks from the 16MB boundary upward. As with ordinary extended memory, HIMEM has to hook the interrupt in order to change the result of this int 15h function, thereby making it look as if the memory is not present in the form expected (since it will now be disbursed to XMS clients). A **/no** command-line switch (perhaps an abbreviation for /nocompaq) exists to disable this provision.

A less happy outcome attends the AT&T 6300 Plus, whose possible configuration with 384KB immediately below the 16MB ceiling of 80286 address space was known in the days when RAMDRIVE.SYS arranged its own extended memory access (the generally undocumented **/u** switch directed RAMDRIVE to build its "disk" in this memory specifically). It seems that the memory cannot be hidden from subsequent programs and cannot therefore be added to the XMS pool with complete safety. Worse, some defensive measure may be advisable because the value returned by int 15h function 88h includes the 384KB even though it is disjoint from the memory installed at 1MB.

Shadow RAM

In many cases of interest, noncontiguous memory is mapped from elsewhere in the CPU address space—from the 384KB region at the top of the first megabyte, where it might otherwise be wasted in space usually reserved for ROM and video RAM. Depending on the hardware, all or some of the redundant RAM in this region may be used to shadow slow ROMs or may be relocated as extended memory. All is well if the relocation is just an addition to the memory already fitted at 1MB (with a corresponding adjustment to the BIOS int 15h function 88h performed as part of BIOS initialization), but if the remapped memory is non-contiguous, then joining it to the XMS pool requires explicit support from the XMS server.

Apart from examples like the AT&T 6300 Plus that predate acceptance of the XMS, the need to meet prior claims to the extra memory can reasonably be set aside because alternative uses for the non-contiguous memory arise only when the memory is mapped elsewhere; as long as the memory is actually addressable as extended memory (determined perhaps by consulting a CMOS configuration or by testing whether data written to the relevant addresses can be read back), its inclusion in the XMS pool should be safe. For instance, this seems intended to happen with HIMEM when running on a Zenith computer that may offer 256KB beginning 384KB below the 16MB boundary. In this particular case, HIMEM would test for the memory's accessibilty just underneath 16MB, add it to the XMS pool, and fill the memory with zeros, but the code built into HIMEM for this purpose confuses some internal flags used by HIMEM to track XMS allocations and does not actually work.

An XMS server may also offer the opportunity to perform the relocation itself, for example, to disable shadow RAM and add the extra memory to the XMS pool. The standard HIMEM.SYS supports only the Compaq scheme in which 128K is mapped from an address immediately below 16MB to segment E000h, where it receives a copy of the video ROM from segment C000h. Since the video ROM and its RAM shadow occupy different addresses, certain relevant interrupt vectors (10h, 1Fh, 43h, and possibly 6Dh) are redirected for shadowing and must be restored to the video ROM when shadowing is disabled—hoping, of course, that no intervening program is relying on interrupt vector addresses recorded while they pointed into segment E000h. Restoration of Compaq's shadow RAM to its place as noncontiguous extended memory is attempted by default if less than 384KB of contiguous extended memory is present at HIMEM's initialization but otherwise requires specification through the **/shadowram:off** switch. Presumably, manufacturers other than Compaq are encouraged to adapt HIMEM for distribution with their machines if disabled shadow RAM is mapped noncontiguously.

Memory Above 64MB

Memory may also lie outside the int 15h scheme simply because too much of it has been installed. Int 15h function 88h returns a 16-bit number of kilobytes and can never indicate the presence of more than 64MB of extended memory. There cannot be many PCs for which this is yet a problem, and, indeed, versions 2.xx of HIMEM.SYS disregard all memory above the 80286 limit of 16MB.

Ultimately, the view to bigger machines demands recognition of more powerful system architectures. The EISA specification provides for flexible configuration of hardware, including memory. Software may interrogate up to sixteen "slots" via int 15h function D800h (EISA Read Slot Configuration Information) and then for each slot, some number of "device functions" via int 15h function D801h (EISA Read Function Configuration Information). This interface is capable of reporting the presence of memory blocks lying at 256-byte boundaries all the way to the end of a 32-bit address space. Each block must be an integral number of kilobytes in size, up to (but not including) a maximum of 64MB (which is not at all restrictive, since a combination of slot and device function may be associated with more than one block).

Immediately after initializing the tables in which it will keep all its information about extended memory (discussed in the next chapter), HIMEM interrogates the EISA slots. All suitable blocks of extended memory found through the EISA interface are added to the pool that HIMEM will distribute among XMS clients. The XMS constrains memory allocations to lie on kilobyte boundaries, so some rounding is required if the EISA reveals a memory region starting on a 256-byte boundary that is not also a kilobyte boundary. Also, the EISA interface may describe a block of memory beginning in the first megabyte, which would be of no interest to HIMEM; such blocks are not added to the XMS pool. A coding oversight in versions 3.xx means that this exception applies to any EISA memory block beginning within a megabyte of any 64MB boundary. Versions 2.xx of HIMEM also reject EISA blocks starting above 16MB (but it seems that these versions of HIMEM will accept EISA memory that straddles the 16MB boundary, as long as it is disjoint from memory beginning at 1MB).

After the EISA slots have been examined, the problem that the EISA interface may duplicate results obtained from int 15h function 88h remains. In its versions 2.xx, HIMEM assumes that the primitive BIOS function will correctly report all contiguous memory extending from the 1MB boundary upward. Since HIMEM coalesces adjacent blocks when adding them to its pool, it suffices now for HIMEM to look through its tables for an entry describing a region that starts at 1MB and then to remove this entry. This way, the EISA interface serves solely to locate non-contiguous memory and does not interfere with HIMEM's measures for compatibility with int 15h allocators over contiguous memory.

With HIMEM versions 3.xx, there is the complication that the EISA region may be larger than the 64MB that the BIOS function is capable of describing. If so, HIMEM treats the surplus as non-contiguous memory but surrenders the first 64MB to be treated compatibly with int 15h allocators.

The real problem is surely that the EISA may indicate the existence of more memory contiguous to 1MB than does int 15h function 88h. The default behavior risks losing track of memory beyond the block described by the BIOS function but below 65MB. Starting with version 3.07, HIMEM offers the switch **/eisa** to ensure that any surplus is measured with respect to the result of int 15h function 88h. It is not clear why this treatment requires a special switch and is not the default.

CHAPTER 11

Disbursing Extended Memory

This chapter begins the process (which will continue over the next several chapters) of building an XMSWATCH utility. Even though enumeration of extended-memory blocks (EMBs) is not part of the XMS, XMSWATCH can list both free and allocated EMBs. Listing allocated EMBs requires that XMSWATCH hook into the XMS chain; this is discussed in Chapter 14.

To build a map of extended memory that is presently free, a program may issue repeated calls to XMS functions 08h and 09h, allocating all free memory from the largest block to the smallest. Information about these blocks is put in structures defined in XMSWATCH.H. Routines in LIST.C arrange for these structures to be listed in ascending order of base address. One benefit of having these routines in their own file is that they can be used again in Chapter 14 when the much harder task of mapping previously-allocated extended memory is tackled. For the moment, FREE.C contents itself with just the memory that is presently free; the main routine is ReportFreeEMBs, which will be called by the top-level module for the overall XMSWATCH utility.

First, however, a brief digression on the High Memory Area (HMA) is necessary.

The HMA

XMSWATCH also reports on HMA usage. The following module determines whether the HMA exists, is in use, or is free. If the HMA is free, HMA.C uses a bisection algorithm to deduce the HMAMIN parameter (such as determined by the HIMEM.SYS **/hmamin** switch or by similar settings with other XMS servers such as QEMM and 386MAX), which XMS clients cannot determine non-invasively. Because an XMS server refuses the HMA to any caller that does not ask for at least HMAMIN bytes, it is possible to binary search for HMAMIN as the boundary between `AllocateHMA` calls that fail and those that succeed. This bisection technique should be familiar to readers from GETLIMIT.ASM in Chapter 8.

```
/* hma.c */

#include  <stdio.h>

#include  <standard.h>
```

```
/* The XMS calling point - already determined by a higher-level module. */
extern void (_far *lpXMS) (void);                    // in main.c

/* In-line definitions for relevant XMS functions - created as macros */

#define GetHMAStatus() \
  HIGHWORD (((DWORD (_fastcall _far *) (WORD)) lpXMS) (0x0000))
#define AllocateHMA(bytes) \
  ((BOOL (_fastcall _far *) (WORD, WORD)) lpXMS) (0x0100, bytes)
#define ReleaseHMA() ((BOOL (_fastcall _far *) (WORD)) lpXMS) (0x0200)

/* A function to determine the current value of the HMAMIN parameter. */
static WORD GetHMAMIN (void);

/* ====================================================================== */

void ReportHMA (void)
{
  printf ("\nThe HMA ");
  if (NOT GetHMAStatus ()) printf ("does not exist for XMS purposes.");
  else if (NOT AllocateHMA (0xFFFF)) printf ("has already been allocated.");
  else {
        ReleaseHMA ();
        printf ("is free - HMAMIN is %u bytes.", GetHMAMIN ());
  }
  printf ("\n");
}

static WORD GetHMAMIN (void)
{
  register WORD big_enough = 0xFFFF, test;
  WORD too_small = 0, step = 0;

  /*    Find HMAMIN by trial and error. The XMS server refuses the HMA to
        all callers who do not claim the need for at least HMAMIN bytes.
        Assuming that the HMA is free when this function is called, then
        AllocateHMA (0) may fail and AllocateHMA (0xFFFF) is guaranteed to
        succeed. Somewhere between these two extremes is the value sought. */

  do {
        if (AllocateHMA (test = too_small + step)) {
          ReleaseHMA ();
          big_enough = test;
        }
        else too_small = test;
  } while (step = big_enough - too_small >> 1);
  return (big_enough);
}
```

In-line XMS Calls from C

Rather than define a C function to represent each XMS function, the Microsoft C compiler provides a `_fastcall` mechanism that can be used to get in-line XMS function calls. As seen in XMSFUNC.H, functions declared with the `_fastcall` keyword place their arguments in the order AX, DX, BX. This is perfect for representing almost all XMS function calls. The lpXMS function pointer returned by int 2Fh function 4310h can simply be cast to the appropriate type of _fastcall function, and the compiler will place the arguments in the appropriate registers. For the HMA and A20 functions, this is not terribly pressing, but extended memory blocks will be manipulated by several modules later in this chapter and in Chapters 13 and 15.

```
/* xmsfunc.h - a collection of macros representing XMS functions */

#include  <standard.h>

/* This header file provides macros that translate the XMS functions as
  in-line C functions. It is assumed that the XMS calling point is
  available as lpXMS. */

extern void (_far *lpXMS) (void);

/* The scheme works because functions declared with the _fastcall keyword
  place their arguments in the order AX, DX, BX. Many XMS functions
  return a Boolean value in register ax. For most practical purposes, the
  error code in bl if ax is returned FALSE is unimportant. Some functions
  return information in dx, where it will be available as the high word of
  a dword function value. */

typedef BOOL _fastcall _far ONEARG_BOOLFUNC (WORD);
typedef BOOL _fastcall _far TWOARGS_BOOLFUNC (WORD, WORD);
typedef BOOL _fastcall _far THREEARGS_BOOLFUNC (WORD, WORD, WORD);

typedef DWORD _fastcall _far ONEARG_DWORDFUNC (WORD);
typedef DWORD _fastcall _far TWOARGS_DWORDFUNC (WORD, WORD);

/* XMS functions 00h, 0Ch, 0Eh, and 10h return values in bx and cannot be
  represented by macros. XMS function 0Bh takes input from DS:SI and
  cannot be represented by a macro. */
```

```
#define AllocateHMA(bytes) ((TWOARGS_BOOLFUNC *) lpXMS) (0x0100, bytes)
#define ReleaseHMA()          ((ONEARG_BOOLFUNC *) lpXMS) (0x0200)

#define GlobalEnableA20()  ((ONEARG_BOOLFUNC *) lpXMS) (0x0300)
#define GlobalDisableA20() ((ONEARG_BOOLFUNC *) lpXMS) (0x0400)
#define LocalEnableA20()   ((ONEARG_BOOLFUNC *) lpXMS) (0x0500)
#define LocalDisableA20()  ((ONEARG_BOOLFUNC *) lpXMS) (0x0600)
#define QueryA20()        ((ONEARG_BOOLFUNC *) lpXMS) (0x0700)

#define QueryFreeXMS()         ((ONEARG_DWORDFUNC *) lpXMS) (0x0800)
#define AllocateEMB(kb)   ((TWOARGS_DWORDFUNC *) lpXMS) (0x0900, kb)
#define ReleaseEMB(handle) ((TWOARGS_BOOLFUNC *) lpXMS) (0x0A00, handle)

#define UnlockEMB(handle)  ((TWOARGS_BOOLFUNC *) lpXMS) (0x0D00, handle)

#define ResizeEMB(handle,newkb) \
  ((THREEARGS_BOOLFUNC *) lpXMS) (0x0F00, handle, newkb)

/* EMB Information structure for XMS function 0Eh */

struct EMBINFO {
  BYTE cFreeHandles;
  BYTE cLocks;
  WORD wSizeInK;
};
```

Two XMS functions that we will need cannot be represented by in-line fastcall macros; these must be clothed as C functions and are provided in XMSFUNC.C:

```
/* xmsfunc.c */

#include    <standard.h>

#include    "xmsfunc.h"

extern void (_far *lpXMS) (void);

/*  Two C-language equivalents for XMS functions 0Ch and 0Eh, which cannot
    be represented by macros using a _fastcall cast because they return
    values in register bx as well as ax and dx. */

BOOL LockEMB (WORD handle, DWORD *baseptr)
{
    DWORD base;
    _asm {
            mov     dx,handle
            mov     ah,0Ch
            call   lpXMS
            or      ax,ax
            jz      done
            mov     word ptr base,bx
            mov     word ptr base+2,dx
    }
    *baseptr = base;
    _asm {
            mov     ax,0001h
        done:
    }
}

BOOL GetEMBInfo (WORD handle, register struct EMBINFO *infoptr)
{
    struct EMBINFO info;
    _asm {
            mov     dx,handle
            mov     ah,0Eh
            call   lpXMS
            or      ax,ax
            jz      done
            mov     info.cFreeHandles,bl
            mov     info.cLocks,bh
            mov     info.wSizeInK,dx
    }
    *infoptr = info;
    _asm {
            mov     ax,0001h
        done:
    }
}
```

Extended Memory Blocks

The following modules supply the easy part of the XMSWATCH utility: to show available extended memory. Similar to what was used for DOS memory blocksin Chapter 4, XMSWATCH will build a list of free extended memory blocks, sorted in ascending order of base address. XMSWATCH.H supplies structures and definitions for analyzing XMS memory allocation.

```
/* xmswatch.h */

#include  <standard.h>
```

```
#pragma pack (1)

/* Structure returned by resident code that monitors XMS activity to record
  handles as they are allocated, allowing non-resident code to ask the
  XMS server for details later. */

struct XMSWATCH {
  WORD wVersion;
  WORD wHandles;
  WORD wHandleTableSize;
  void (_far *lpNextXMS) (void);
};

#define XMSWATCH_VERSION  (MK_WORD (1, 00))

/* Information about extended memory blocks is built into a list of HANDLE
  structures (linked in increasing order of linear address). */

struct HANDLE {
  struct HANDLE *pNext;
  WORD wFlags;
  WORD wHandle;
  DWORD dwBase;
  WORD wSizeInK;
  BYTE cLocks;
  CHAR cOwner [9];   // eight bytes for name, plus null terminator
};

/* Bits for the flag word */

#define POOL_HANDLE             0x0001    // only one of
#define FREE_EMB             0x0002     // these three
#define USED_EMB             0x0004     // may be set

#define PHYSICAL_EMB         0x0010

#define ENTRY_USED 0x8000

/* The HANDLE structures are arranged in an array immediately after a few
  fields that help with list management. */

struct LIST {
  struct HANDLE *pLowest;
  struct HANDLE *pUnused;
  struct HANDLE *pEnd;
  struct HANDLE array [];
};

#pragma pack ()

/* Functions provided in LIST.C for list management */

struct LIST *CreateList (WORD);
struct HANDLE *GetUnusedEntry (struct LIST *);
struct HANDLE *AddEntry (struct LIST *, DWORD);
void LinkEntry (struct LIST *, struct HANDLE *);
```

LIST.C contains routines for constructing and maintaining an ordered, linked list of entries describing XMS allocations.

```
/* list.c */

#include  <malloc.h>
#include  <memory.h>

#include  <standard.h>

#include  "xmswatch.h"

struct LIST *CreateList (WORD handles)
{
  register struct LIST *list;
  WORD space;

  /*      The memory requirement is the given number of HANDLE structures,
          plus the few fields in the list header. Allocate the necessary
          memory from the heap, initialize the header, and clear the space
          occupied by the HANDLE structures. */

  space = handles * sizeof (struct HANDLE);
  list = (struct LIST *) malloc (sizeof (struct LIST) + space);
  if (list != NULL) {
          list -> pLowest = NULL;
          list -> pUnused = list -> array;
          list -> pEnd = list -> pUnused + handles;
          memset (list -> array, 0, space);
  }
  return (list);
}
```

```
struct HANDLE *GetUnusedEntry (register struct LIST *list)
{
  register struct HANDLE *ptr;

  /*      Advance through the array of HANDLE structures, looking for one that
          has not been marked as used. Keep the pUnused field in the LIST
          header as a 'bookmark' for the next time the function is called. */

  ptr = list -> pUnused;
  while (ptr < list -> pEnd) {
          if (NOT (ptr -> wFlags & ENTRY_USED)) {
            ptr -> wFlags = ENTRY_USED;
            list -> pUnused = ptr + 1;
            return (ptr);
          }
          ptr ++;
  }
  ptr = list -> array;
  while (ptr < list -> pUnused) {
          if (NOT (ptr -> wFlags & ENTRY_USED)) {
            ptr -> wFlags = ENTRY_USED;
            list -> pUnused = ptr + 1;
            return (ptr);
          }
          ptr ++;
  }
  return (NULL);
}

/* Given the base address for an EMB that is not yet listed, the following
  function assigns the EMB to an unused HANDLE structure in the list and
  connects the new entry to the existing chain. */

struct HANDLE *AddEntry (struct LIST *list, DWORD newbase)
{
  register struct HANDLE **mark, *ptr;
  struct HANDLE *unused;

  /* Claim an unused entry. */
  unused = GetUnusedEntry (list);
  if (unused == NULL) return (NULL);

  /*      Entries are ordered according to the linear address of the EMBs that
          they describe. Advance the ptr variable from the entry for the
          lowest EMB until it describes an EMB with higher base address than
```

```
        the argument newbase. Before each advance, save the address of the
        pNext field. This will be the insertion point for a new entry when
        the search concludes. */

  mark = &(list -> pLowest);
  ptr = list -> pLowest;
  while (ptr != NULL AND ptr -> dwBase <= newbase) {
        mark = &(ptr -> pNext);
        ptr = ptr -> pNext;
  }

  /*    Thread the next unused HANDLE entry into the list between the
        entries pointed to by mark (indirectly) and ptr. */

  unused -> pNext = ptr;
  *mark = unused;

  /*    Fill in the linear address field. */
  unused -> dwBase = newbase;

  /*    Return the address of the newly-assigned HANDLE structure so that
        the caller may enter additional information. */

  return (unused);
}

/* The next function is similar to the one above but works with a HANDLE
  structure that has already been initialised and need only be linked with
  the other entries in the list. */
void LinkEntry (struct LIST *list, struct HANDLE *entry)
{
  register struct HANDLE **mark, *ptr;
  DWORD newbase = entry -> dwBase;

  mark = &(list -> pLowest);
  ptr = list -> pLowest;
  while (ptr != NULL AND ptr -> dwBase <= newbase) {
        mark = &(ptr -> pNext);
        ptr = ptr -> pNext;
  }

  entry -> wFlags |= ENTRY_USED;
  entry -> pNext = ptr;
  *mark = entry;
}
```

FREE.C provides the routine `ReportFreeEMBs`, which XMSWATCH will use to display an ordered list of free extended memory blocks. The list itself is created by `BuildListOfFreeEMBs`. XMS provides very little informatation about free extended memory: just the total amount available and the size of the largest free block. To discover the actual distribution of free extended-memory blocks, `BuildListOfFreeEMBs` must temporarily allocate each such block, working from the largest downward.

```
/* free.c */

#include  <stdio.h>

#include  <standard.h>

#include  "xmsfunc.h"
#include  "xmswatch.h"

/* XMS services (the familiar pointer to the XMS calling address, some
  macros that serve as in-line functions and some genuine functions) */

extern void (_far *lpXMS) (void);

extern BOOL LockEMB (WORD, DWORD *);                      // in xmsfunc.c
extern BOOL GetEMBInfo (WORD, struct EMBINFO *);          // in xmsfunc.c

BOOL AllocateLargestEMB (WORD *, WORD *);
WORD QueryFreeHandles (void);

/* A pointer and function that assist the management of data about free
  memory blocks and unassigned handles */

struct LIST *pFreeEMBs;
WORD BuildListOfFreeEMBs (struct LIST *);

/* In addition to various sentences, the report contains a table with two
  columns under the following header. */

static CHAR header [] = "\n\
 Linear Address Size (KB) \n\
 --------------------------";

/* Functions that deal with the presentation of particular items of data */
void PutLinearAddress (struct HANDLE *);
void PutSizeInK (struct HANDLE *);
void PutShortfall (WORD);
```

```
/* ================================================================== */

void ReportFreeEMBs (void)
{
  register struct HANDLE *ptr;
  register WORD handles;
  WORD unaccounted;

  printf ("\nReport on Free Extended Memory:");

  /*      Begin with the total amount of free extended memory managed through
          the XMS. This is returned in dx by the XMS function that provides
          information about extended memory availability. */

  printf ("\n\n Free memory (KB): %u", HIGHWORD (QueryFreeXMS ()));

  /*      Find out how many XMS handles are available for association with
          extended memory blocks. */

  handles = QueryFreeHandles ();
  printf ("\n Available handles: %u\n", handles);
  if (handles == 0) return;

  /*      The XMS provides only a little information about free extended
          memory: the total amount available and the largest free block. To
          discover the distribution of free blocks requires that each such
          block be allocated temporarily, working from the largest downward.
          Do not display information until the complete picture has been
          uncovered. Instead, build the results into a sorted list so they
          may be presented in order of increasing base address rather than
          decreasing size.

          Note that the BuildListOfFreeEMBs function releases all the
          temporary allocations once it has completed its data collection, so
          the list is just a snapshot of the situation sometime during the
          execution of that function.

          One side-effect of delaying the report is that no DOS functions are
          called while this program has allocated XMS handles. This avoids
          complications with DOS exceptions that might terminate the program
          before the XMS handles have been returned to the system. */

  pFreeEMBs = CreateList (handles);
  if (pFreeEMBs == NULL) {
          printf ("\nInsufficient internal heap space to complete analysis\n");
```

```
        return;
  }

  unaccounted = BuildListOfFreeEMBs (pFreeEMBs);

  /*      If the only entries in the table describe handles taken from the
          pool, do not present an empty table of results. In this case, the
          list has been built only to obtain the numerical values of the spare
          handles (a meaningful notion only for some XMS servers) should they
          be needed by a future extension to the program. */

  ptr = pFreeEMBs -> pLowest;
  for (;;) {
          if (ptr == NULL) return;
          if (NOT (ptr -> wFlags & POOL_HANDLE)) break;
          ptr = ptr -> pNext;
  }

  /*      Retrieve the results from the list, following the links that have
          been set up in ascending order of base address. */

  printf (header);

  for (ptr = pFreeEMBs -> pLowest; ptr != NULL; ptr = ptr -> pNext) {
          /* For each handle associated with a non-empty block of memory,
            display the base address and size, separating them with enough
            spaces to fit correctly in the columns defined by the header. */

          if (NOT (ptr -> wFlags & POOL_HANDLE)) {
            printf ("\n   ");
            PutLinearAddress (ptr);
            printf ("   ");
            PutSizeInK (ptr);
          }
  }
  if (unaccounted) PutShortfall (unaccounted);
  printf ("\n");
}

void PutLinearAddress (struct HANDLE *ptr)
{
  /*      If the EMB described by the designated HANDLE structure could not be
          locked, display a set of question marks to indicate that the base
          address is unknown. */
  if (ptr -> wFlags & PHYSICAL_EMB) printf ("%08lXh", ptr -> dwBase);
  else printf ("???????? ");
}

void PutSizeInK (struct HANDLE *ptr)
{
  printf ("%5u", ptr -> wSizeInK);
}

void PutShortfall (WORD amount)
{
  printf ("\n\n Unable to account for additional %uK", amount);
}

WORD BuildListOfFreeEMBs (register struct LIST *list)
{
    register struct HANDLE *ptr;
    WORD handle;
    WORD sizeinkb, unaccounted;
    DWORD linear_address;

    /*    Keep allocating XMS handles, working downwards from the largest
          free block. */

    while (AllocateLargestEMB (&handle, &sizeinkb)) {

          /* Lock the EMB in order to obtain its linear address. Obtain an
             entry for this block in the designated list, using the base
             address as the sort key. A lock failure presumably indicates
             virtual memory. Place such blocks at the end of the list. */

          if (LockEMB (handle, &linear_address)) {
              ptr = AddEntry (list, linear_address);
              if (ptr == NULL) break;                 // shouldn't happen
              ptr -> wFlags |= PHYSICAL_EMB;          //     but check anyway
          }
          else {
              ptr = AddEntry (list, -1);
              if (ptr == NULL) break;
          }

          /* The AddEntry function records the block's linear address in the
             appropriate field. Since it has already used the base address
             to find a suitable place in the list, it may as well also store
             the value. The handle and size, however, still need to be
```

```
            entered, along with flags that describe the block's status. */

        ptr -> wHandle = handle;
        ptr -> wSizeInK = sizeinkb;
        ptr -> wFlags |= (sizeinkb ? FREE_EMB : POOL_HANDLE);
    }

    /*  Check to see that the handle supply was not exhausted before all
        extended memory had been allocated. */

    unaccounted = HIGHWORD (QueryFreeXMS ());

    /*  Tidy up. All the XMS handles that have been allocated just to
        discover the layout of free extended memory must be unlocked (if
        necessary) and returned to the system. */

    for (ptr = list -> pLowest; ptr != NULL; ptr = ptr -> pNext) {
        if (ptr -> wFlags & (FREE_EMB | POOL_HANDLE)) {
            handle = ptr -> wHandle;
            if (ptr -> wFlags & PHYSICAL_EMB) UnlockEMB (handle);
            ReleaseEMB (handle);
        }
    }

    /*  Return the amount of extended memory that could not be allocated. */
    return (unaccounted);
}

BOOL AllocateLargestEMB (WORD *handleptr, WORD *sizeptr)
{
    register WORD sizeinkb;
    DWORD temp;
    struct EMBINFO info;

    /*  Determine the size of the largest free EMB, then attempt to allocate
        it. */

    sizeinkb = (WORD) QueryFreeXMS ();
    temp = AllocateEMB (sizeinkb);
    if ((BOOL) temp) {

        /* If successful, record the handle (returned in register dx by the
           allocation function and subsequently stored as the high word of
           the temp variable). */
        *handleptr = HIGHWORD (temp);

        /*  As a defensive measure, use XMS function 0Eh to check the EMB
            size. It is actually possible for HIMEM.SYS to allocate a
            larger block than requested (in an admittedly exceptional
            circumstance where the allocation happens to exhaust the supply
            of spare handles). */

        *sizeptr = GetEMBInfo (*handleptr, &info) ? info.wSizeInK : sizeinkb;
        return (TRUE);
    }
    return (FALSE);
}

WORD QueryFreeHandles (void)
{
    register WORD freehandles = 0;
    register WORD dummyhandle;
    DWORD temp;
    struct EMBINFO info;

    /*   Allocate a dummy handle so that XMS function 0Eh can be used to
         obtain the number of other handles available. */

    temp = AllocateEMB (0);
    if ((BOOL) temp) {
         dummyhandle = HIGHWORD (temp);
         if (GetEMBInfo (dummyhandle, &info)) freehandles = info.cFreeHandles;

         /*  Release the dummy handle and adjust the count of available
             handles accordingly. */
         if (ReleaseEMB (dummyhandle)) freehandles ++;
    }
    return (freehandles);
}
```

Because FREE.C must temporarily allocate each free EMB, it requires a certain number of EMB handles. The XMS NUMHANDLES setting can generally be changed by a user with the HIMEM.SYS **/numhandles** switch or with similar options such as XMSHNDL in 386MAX. However, as with the HMAMIN setting, there is no non-intrusive way for a program to discover this crucial setting. The numhandles setting is particularly important, because, as discussed in further detail in Chapter 12, the XMS allocation and resize functions depend on a pool of free handles to manage newly-freed blocks.

DOS 6 HIMEM Facility for Enumerating EMBs

Since we have just developed our own facility for enumerating XMS handles, it is important to note that, starting with HIMEM.SYS version 3.09 (February 1993; distributed with DOS 6), there is an int 2Fh function 4309h to query HIMEM's array of handle descriptors. MEM.EXE in DOS 6 uses this function.

Table 11-1: Handle Descriptor Implemented by HIMEM.SYS Versions 3.07 and 3.09

OFFSET	SIZE	DESCRIPTION	
00h	byte	status	
		01h	handle associated with free EMB
		02h	handle associated with allocated EMB
		04h	handle in pool - not associated with any EMB
01h	byte	lock count	
02h	dword	linear address of EMB, measured in KB	
06h	dword	size of EMB, measured in KB	

Some HIMEM.SYS versions that pre-date XMS 3.0 allow only 16-bit values for the address and size of an EMB (and therefore have a six-byte handle descriptor). This is compatible with the design limit of 64MB for EMBs in version 2.0 of the XMS, but restricts support to blocks with base addresses lying in the first 64MB of memory. This is no concern to many HIMEM.SYS variants, which ignore memory above the 16MB limit of the 80286. Some (most notably, Compaq's version 2.66) also implement the ten-byte handle descriptor so that they may support memory throughout the 80386 address space, though admittedly still constrained to 64MB in any single block.

The DOS 6 version of MEM.EXE requires the specific value 01h in the first field of the structure in Table 11-2 below and insists on the use of ten-byte handle descriptors.

While useful, this belated addition to HIMEM.SYS of functionality to enumerate XMS handles does not eliminate the need for an XMSWATCH utility. First, note that int 2Fh function 4309h is a HIMEM function, *not* an XMS function. For older versions of HIMEM and for other primary XMS servers such as QEMM386 or 386MAX, to say nothing of the XMS emulation in the Windows V86MMGR VxD (which is not primary, but since it hardly passes anything down the chain, it may as well be), readers will need something like XMSWATCH to find out what's been going on in extended memory.

Table 11-2: Int 2Fh Support for Direct Inspection of HIMEM's EMB Handles

INPUT:

AX 4309h

OUTPUT:

al 43h

es:bx address of structure describing HIMEM's array of handle descriptors:

OFFSET	SIZE	DESCRIPTION
00h	byte	01h
01h	byte	size of handle descriptor
02h	word	number of handles
04h	dword	address of handle descriptor array

CHAPTER 12

HIMEM.SYS at Close Range

Throughout the preceding chapters the behavior of HIMEM.SYS has been used to illustrate the practical considerations that apply to the design of an XMS server, particularly one that assumes control of extended memory in real mode and continues to operate in real mode. HIMEM.SYS is the standard XMS implementation and even readers who have replaced it with another need to understand its behavior.

Command-Line Configuration

Although exceptions are made for some switch arguments, in general HIMEM.SYS assumes the command line seen at device-driver initialization to have been capitalized (which may have consequences for programs that load device drivers after system startup) and to be terminated by a carriage-return or line-feed. Only spaces are regarded as separators (which may create a small mystery if a text editor puts tabs in the CONFIG.SYS file).

Switches must be introduced by the '/' character and are specified by the character immediately afterward. Further characters are irrelevant up to a ':' to introduce a string or an '=' for a numerical value. Recent versions of HIMEM accept simple switches that have no variable component. A summary of the recognized switches is given in Table 12-1. HIMEM treats boolean arguments as strings starting with the characters 'ON' and 'OFF' as if to check only the first two letters of an ON|OFF choice. String arguments longer than 12 bytes overflow the space allowed (but the longest accepted machine name—att6300plus—is 11 bytes, so ordinarily there will be no problem). Switches may be repeated.

Of the allowed switches, the actions of all but two have been explained in earlier chapters. The undocumented **/z** switch is discussed in Chapter 14 because its role becomes non-trivial only when additional XMS servers have hooked into HIMEM's XMS interface. The **/verbose** switch is entirely a user-level concern with no consequences for programmers after HIMEM's initialization. It counteracts the default behavior of HIMEM.SYS version 3.09, which is to provide no status information during startup. Thus, users who like to see their device drivers' startup messages, if only for assurance after a spectacularly bad crash during program development, must now add this switch to HIMEM's **device** line in CONFIG.SYS or hold down the Alt key while the system initializes.

Table 12-1: Command-Line Options for HIMEM.SYS Versions 2.77, 3.07, and 3.09 (Released with DOS 5, Windows 3.1, and DOS 6, Respectively)

SWITCH	RANGE AND AVAILABILITY	DEFAULT
/A:Boolean		ON
/C:Boolean		OFF
/E	version 3.07 and higher	
/H=n	n ≤ 64	0
/I=n	64 ≤ n ≤ xmem	0
/M:*name*	*name* from Table 9-1 in Chapter 9; or equivalent number	
/N=*n*	0 < *n* ≤ 128	32
/NO	version 3.09 only	
/S:*Boolean*	Compaq machines only	OFF if *xmem* < 384KB; ON if *xmem* ≥ 384KB
/V	version 3.09 only	
/Z	version 3.09 only	

where xmem *(in KB) denotes the amount of installed extended memory reported by int 15h function 88h—up to a maximum of 15MB in HIMEM versions 2.xx, which ignore any memory above the 16MB boundary of the 80286 address space. (Memory above 16MB is sometimes called "Super Extended Memory.")*

The provision for displaying status information when the user requests it specifically is an arguably good compromise. This behavior is offered by the DOS 6 versions of both HIMEM.SYS and EMM386.EXE, so it appears to be some sort of standard that new device drivers may care to follow.

XMS Implementation

At the programming level, the distinction between a general interface specification and a particular implementation may occasionally become important. Ideally, programmers could work to the published interface and only developers of XMS servers would concern themselves with the details of implementation. It is often easier, however, to understand abstract notions after examining a concrete representation. With HIMEM.SYS and the XMS, there may also be the suspicion that the general interface has been extrapolated from the particular anyway.

Given that the XMS specification is available in its "pure" form free of charge from Microsoft and that little would be gained by recycling it here, the author has instead chosen to present the specification as actually implemented by HIMEM. It is not suggested that programmers rely on features specific to HIMEM's implementation of the XMS, but differences between the general and the particular may warrant defensive programming. For example, HIMEM relocates an EMB only on receipt of implicit permission from the XMS caller (through function 0Fh) and therefore a physical address obtained by locking an EMB is not invalidated by releasing the lock. A programmer would be unwise to rely on this behavior (as does the initialization code of the EMM386 program distributed with DOS 5), but on the other hand, a programmer almost certainly ought to be aware that calls to HIMEM may affect registers that the XMS documentation states should be preserved. As another example, the most widely available specification is for XMS 2, but HIMEM.SYS now implements XMS 3.

Incidentally, outdated but still marginally useful source code for HIMEM.SYS version 2.01 (July 1988) is available on CompuServe (see HIMEM.ARC in IBMPRO/OS Services and IBMSYS/DOS Utilities). This file was "uploaded with Microsoft's blessing by Steve Gibson, InfoWorld." Within HIMEM.ARC, SOURCE.ARC contains HIMEM.ASM and XM286.ASM. HIMEM.ASM contains the following comment, of possibly historical interest:

```
;* The HMA was originally envisioned by Ralph Lipe.
;* Original XMS spec designed by Aaron Reynolds, Chip Anderson, and
;* Tony Gosling. Additional spec suggestions by Phil Barrett and David
;* Weise of Microsoft, Ed McNierney of Lotus and Bob Smith of Qualitas.
;*
;* MoveExtMemory function written by Tony Gosling.
```

HIMEM supports functions 00h to 0Fh inclusive, together with some few variations described in the next paragraph. All others return AX = 0000h and BL = 80h (function not implemented).

In versions 3.xx, functions may be called with the high bit set in AH to indicate use of the 32-bit interface. This high bit is meaningful only for functions 08h, 09h, 0Eh, and 0Fh—in all other cases, it is ignored. On 80286 machines, calling HIMEM with AH = 88h, 89h, 8Eh, or 8Fh seems intended to result in the return of AX = 0000h and BL = 80h (function not implemented). The more likely result is a system crash because the relevant code is called as a FAR procedure but attempts a NEAR return.

In versions 2.xx, only function 0Bh has separate code for the 80286 and 80386 processors. Versions 3.xx extend this to functions 08h to 0Bh, 0Eh, and 0Fh (and their 32-bit variations).

When the system is configured with **dos=high**, a piece of HIMEM may be moved to the high memory area. In this case, some functions cause the A20 to be enabled during processing (although, the A20 state is preserved across the XMS call)—these are 00h and all functions 08h to 0Fh inclusive.

All XMS functions use register AX to return a value—typically a Boolean value. Some functions return information in registers BX and DX if successful. Register BL is always used for the error code when a function fails.

Registers CX, SI, DI, BP, DS, ES, and FLAGS are preserved. It is not observed often enough in discussions of XMS that registers BX and DX should be regarded as corruptible. Microsoft's XMS documentation may state that "XMS drivers must preserve all registers except those containing returned values across any function call," but this is worthless when Microsoft's own implementation goes against the advice. Specifically, for HIMEM.SYS, all functions not intended to return a meaningful value in BX when successful clear BL to zero. Some functions that normally return values in DX clear that register to zero on failure.

Additionally, two circumstances may clear the DX register unexpectedly. These possible error conditions may occur for almost all HIMEM functions and are not included in Table 12.2. If a VDISK driver has prevented complete installation of HIMEM, then all supported functions other than 00h return AX = 0000h, BL = 81h (VDISK allocator detected), DX = 0000h. If part of HIMEM is executing in the HMA, then an A20 failure on attempting to call this code or on returning from it is reported by AX = 0000h, BL = 82h (A20 error), DX = 0000h, even if the body of the relevant function has been processed successfully.

Table 12-2: XMS Functions Implemented by HIMEM.SYS Versions 2.77, 3.07, and 3.09 (Released with DOS 5, Windows 3.1, and DOS 6, Respectively)

HIMEM function 00h—Get XMS Version

INPUT:

AH	00h

OUTPUT:

ax	XMS version in BCD (e.g., 0200h for 2.00)	
bx	HIMEM version in BCD (e.g., 0277h for 2.77)	
dx	HMA existence	
	0000h	if HIMEM does not regard the HMA as present
	0001h	if it does

The notion of HMA existence is changeable. At first, the HMA is held to exist if during HIMEM's installation, int 15h function 88h reported at least 64KB of memory present at the 1MB boundary. HIMEM does not actually assume responsibility for managing contiguous extended memory until some XMS function other than this one is called; at this time, the HMA ceases to exist in HIMEM's reckoning if either the **/int15** switch was used to reserve extended memory from the 1MB boundary upward or int 15h no longer reports the 64K as present.

Some confusion seems to exist over the interpretation of HIMEM version numbers—even among Microsoft's programmers. Observe, for instance, that the Windows 3.0 versions of WIN.COM and DOSX.EXE disallow BX < 0228h and BX < 021Ch, respectively. Since 1Ch equals 28 in decimal, it looks as if both programs intend the same thing, that is, to insist on having HIMEM version 2.28 or better, but the second value cannot possibly be taken as BCD.

Table 12-2: Continued

HIMEM function 01h—Allocate HMA

INPUT:

AH	01h
DX	number of bytes required

SUCCESS:

ax	0001h
bl	00h

FAILURE:

ax	0000h	
bl	90h	if the HMA is not managed by HIMEM
	91h	if the HMA is already allocated
	92h	if the amount requested is less than the size specified by the **/hmamin** switch

If the function is called with AL = 40h, it will also fail (with error code 91h) if DX equals the number of KB of free extended memory recorded by the last call to HIMEM function 08h (but in versions 3.xx, this value is recorded only by the 80286-specific coding of function 08h). The calling value of AL does not feature in the documented XMS interface. Its special meaning to HIMEM may cause the function to fail mysteriously. Chapter 15 discusses how RAMDRIVE and SMARTDRV handle XMS calls with AX = 0140h.

Table 12-2: Continued

HIMEM function 02h—Free HMA

INPUT:

AH	02h

Table 12-2: Continued

SUCCESS:

ax	0001h	
bl	00h	

FAILURE:

ax	0000h	
bl	93h	if the HMA is not yet allocated

HIMEM function 03h—Global Enable A20 Line

INPUT:

AH	03h

SUCCESS:

ax	0001h
bl	00h

FAILURE:

ax	0000h
bl	82h

On first use, this function is equivalent to a Local Enable A20. If this is successful, all subsequent calls—until cancelled by the Global Disable A20 function—succeed trivially.

Table 12-2: Continued

HIMEM function 04h—Global Disable A20 Line

INPUT:

AH	04h

SUCCESS:

ax	0001h
bl	00h

FAILURE:

ax	0000h
bl	82h

Refer to earlier notes for the Global Enable A20 Line function.

Table 12-2: Continued

HIMEM function 05h—Local Enable A20 Line

INPUT:

AH	05h

SUCCESS:

ax	0001h
bl	00h

FAILURE:

ax	0000h
bl	82h

If HIMEM has not installed an A20 handler (because the A20 line was already on during the driver's initialization and the **/a20control:off** switch was given), then both Local A20 functions return success trivially. This case is discounted from further observations about the Local A20 functions.

The driver maintains an internal counter to match Local Enable and Local Disable functions against each other. Successful calls to these functions increment and decrement the counter, respectively. On each call to a Local A20 function (including those implied by other HIMEM functions), the driver reconciles the physical state of the A20 to its internal counter, ensuring that the A20 is disabled when the counter is zero and enabled otherwise. The function returns an error only after three failures to change the physical A20 state.

Table 12-2: Continued

HIMEM function 06h—Local Disable A20 Line

INPUT:

AH	06h

SUCCESS:

ax	0001h
bl	00h

FAILURE:

ax	0000h
bl	82h

Refer to earlier notes for the Local Enable A20 Line function.

Table 12-2: Continued

HIMEM function 07h—Query A20 Line Status

INPUT:

AH	07h

OUTPUT:

ax	0000h if A20 line disabled 0001h if A20 line enabled
bl	00h

The default testing method is to choose eight bytes from among the 128 bytes beginning at addresses 0000:0000 and at FFFF:0010, cycling the testing location eight bytes ahead each time the test is performed. If the eight bytes differ, then clearly the A20 line is enabled. If they are equal (hopefully not by accident), then the A20 line is held to be disabled.

The installed A20 handler may provide its own facility for querying the A20 status. Of the handlers supplied in the HIMEM.SYS file, only those for the ptlcascade, ps2, and ibm7552 offer this feature.

Table 12-2: Continued

HIMEM functions 08h and 88h—Query Free Extended Memory

INPUT:

AH	08h 88h (available with HIMEM versions 3.xx on 80386 processor)

SUCCESS:

(e)ax	size in KB of largest free block
bl	00h
(e)dx	total in KB of all free blocks

FAILURE:

(e)ax	0000h
bl	A0h
(e)dx	0000h

CORRUPTS:

bh

high word of ecx, if AH = 88h

The error condition occurs if no free block exists or if all free blocks are empty (so it is not possible for HIMEM to return zero for ax or dx without also indicating an error).

In versions 3.xx with an 80386 processor or better, function 08h returns the size and total as FFFFh if they are too large to fit in the 16-bit registers.

XMS 3.x function 88h attempts to provide ECX = highest KB managed by HIMEM, but since CX is preserved explicitly by the code that dispatches functions to the responsible procedures, the high word of ECX must be regarded as corrupt.

Table 12-2: Continued

HIMEM functions 09h and 89h—Allocate Extended Memory Block

INPUT:

AH	09h
	89h (available with HIMEM versions 3.xx on an 80386 processor)
(E)DX	requested size in KB (may be zero)

SUCCESS:

ax	0001h
bl	00h
dx	handle

FAILURE:

ax	0000h
bl	A0h or A1h (details below)
dx	0000h

CORRUPTS:

bh

In HIMEM's allocation scheme, each Extended Memory Block (EMB) is associated with a handle, even if the block is free. The converse is not necessarily true, however. In general, there exists a pool of handles not associated with any memory blocks. Allocating a new block requires not only the existence of a free block with sufficient size (and its own handle) *but also* the availability of a handle from the NUMHANDLES pool to take up any unrequired portion of the free block. The behavior of the allocation procedure when this condition is not satisfied will not strike all programmers as straightforward:

Table 12-2: Continued

FREE BLOCK OF SUFFICIENT SIZE	REQUESTED SIZE	SPARE HANDLE IN POOL	RESULTING ACTION
yes		yes	allocation taken from base of free block; remainder made into new free block using handle from pool
yes		no	whole of free block allocated *even if larger than request*
no	DX = 0	yes	handle from pool made into free block of zero size
no	DX ≠ 0	yes	error A0h (memory exhausted)
no		no	error A1h (handles exhausted)

As a result, handles can never be exhausted before all memory has been allocated, but some XMS client may be given more memory than it asked for (or realizes).

The allocation procedure uses the numerically least handle associated with a block of sufficient size.

Table 12-2: Continued

HIMEM function 0Ah—Free Extended Memory Block

INPUT:

AH	0Ah
DX	handle

SUCCESS:

ax	0001h
bl	00h
bh	corrupt
dx	corrupt

FAILURE:

ax	0000h	
bl	A2h	if handle is invalid or unallocated
	ABh	if handle is locked

The handle is returned to the pool if the block with which it was associated had size zero. Otherwise the block is coalesced with any number of free blocks immediately above or below it in memory. All handles except for the one associated with the block lowest in memory are returned to the pool.

Table 12-2 Continued

HIMEM function 0Bh—Move Data to or from Extended Memory Block

INPUT:

AH 0Bh

DS:SI address of parameter block

OFFSET	SIZE	DESCRIPTION
00h	dword	transfer count in bytes
04h	word	source handle
06h	dword	offset into source (or far pointer if source handle is zero)
0Ah	word	destination handle
0Ch	dword	offset into destination (or far pointer if destination handle is zero)

SUCCESS:

ax 0001h

bl 00h

FAILURE:

ax 0000h

bl	82h	if enabling the A20 line for the transfer or disabling it afterward (even if the data has been moved successfully) fails or if protected mode call to int 15h function 87h returns error 03h (also an A20 error)
	A3h	if (non-zero) source handle is unallocated
	A4h	if offset into source exceeds block size
	A5h	if (non-zero) destination handle is unallocated
	A6h	if offset into destination exceeds block size
	A7h	if transfer count is odd or would overrun block's end or if protected mode call to int 15h function 87h returns error 02h (processor exception during data transfer, but often used to indicate a privilege violation)
	A9h	if protected mode call to int 15h function 87h returns error 01h (parity error)

This function is handled with processor-specific code. Only the relevant version is retained in memory when the device driver is installed.

For the 80286, access to extended memory is obtained through the undocumented CPU instruction LOADALL discussed in Chapter 8.

The 80386 version recognizes that segment sizes in that processor's real mode are not restricted to 64KB and traps int 0Dh (for the duration of the extended memory access only) to expand segments to cover the whole 4GB address space if they are not large enough to accommodate all the memory involved in the transfer. This trick is not available if HIMEM is called within a virtual machine, because control of segment descriptors lies with the virtual-8086 monitor in ring 0. Some cooperation is required in this environment and the least restrictive assumption is that the monitor will supply a protected-mode version of the BIOS's int 15h function 87h (which may also settle questions of privilege, make special arrangements for memory-mapped devices, etc.).

For both processors, interrupts are enabled during the processing of this function in real mode. That this be possible is essential to satisfactory performance when accessing extended memory from real mode, but it means that the function is not suitable for use in a section of code for which interrupts must be disabled throughout. EMBs involved in the transfer are locked during all interrupt windows in which transfer code depends on having stable addresses for the source and destination.

Data transfer is performed by REP MOVS instructions. During the transfer, the source and destination pointers are advanced upward through memory, which means that if the regions involved in the move intersect, the function should not be used unless the source is higher than the destination. Overlap can happen only if the source and destination handles are equal.

Table 12-2: Continued

HIMEM function 0Ch—Lock Extended Memory Block

INPUT:

AH	0Ch
DX	handle

SUCCESS:

ax	0001h
dx:bx	physical address of block

FAILURE:

ax	0000h	
bl	A2h	if handle is invalid or unallocated
	ACh	if lock count for handle is already the maximum allowed (FFh)
bh	corrupt	
dx	0000h	

Locking prevents a block from being released with XMS function 0Ah or resized with XMS function 0Fh.

Whether the XMS lock function returns a linear or (as shown previously) a physical address is a matter of some debate. The XMS 3 specification says explicitly that the lock function returns a *physical* address. However, the XMS 2 specification was silent on this point, presumably because there's no distinction for HIMEM (linear == physical) and because Windows enhanced mode dodges the issue by not letting you lock an XMS block. Given that the XMS 2 specification didn't address this issue, it is not surprising that the XMS server in at least one 386 memory manager returned a *linear* address.

The vagueness in XMS 2.0 probably arises from the fact that this specification was an outgrowth of HIMEM.SYS and not the other way around. The canonical XMS server behind the specification was HIMEM rather than a dual XMS server and full-fledged paging memory manager such as 386MAX or QEMM386.

In one context, having lock return a physical address makes no sense. It is very unlikely that the XMS client will be in a position (or have the desire) to access the memory by its physical address. However, a physical address is required for the changeover from one protected mode environment to another. The new system allocates all free extended memory from the old system. The addresses it gets from locking these blocks go into the new system's page tables. They won't be much good unless they're physical addresses.

Table 12-2: Continued

HIMEM function 0Dh—Unlock Extended Memory Block

INPUT:		
AH	0Dh	
DX	handle	
SUCCESS:		
ax	0001h	
bl	00h	
FAILURE:		
ax	0000h	
bl	A2h	if handle is invalid or unallocated
	AAh	if handle is not locked
CORRUPTS:		
bh		

Table 12-2: Continued

HIMEM functions 0Eh and 8Eh—Get Information about Extended Memory Block

INPUT:

AH	0Eh
	8Eh (available with HIMEM versions 3.xx on an 80386 processor)
DX	handle

SUCCESS:

ax	0001h
bh	lock count
bl	number of unallocated handles;
	00h if AH = 8Eh
(e)dx	size of block in K

FAILURE:

ax	0000h	
bl	A2h	if handle is invalid or unallocated;
		or for function 0Eh in versions 3.xx running on an 80386 or better—if the size parameter exceeds FFFFh, i.e., if the block is 64MB or larger

In versions 3.xx (whatever the CPU), function 0Eh returns the number of unallocated handles as FFh if the actual number is too large to fit in register BL. This is irrelevant to present versions of HIMEM.SYS because the number of handles is constrained to a maximum of 128.

Function 8Eh counts the number of unallocated handles (as a word) but does not return the result.

Table 12-2: Continued

HIMEM functions 0Fh and 8Fh—Reallocate Extended Memory Block

INPUT:

AH	0Fh
	8Fh (available with HIMEM versions 3.xx on an 80386 processor)
(E)BX	new block size in K
DX	handle

Table 12-2: Continued

SUCCESS:

ax	0001h
bl	00h

FAILURE:

ax	0000h	
bl	A0h	if new allocation is sought but cannot be satisfied
	A1h	if handle from pool is needed but none are available
	A2h	if handle is invalid or unallocated
	ABh	if handle is locked

CORRUPTS:

bh

dx

high word of eax, if AH = 8Fh

high word of edx, in special circumstance noted below

bp, in special circumstance noted below

The function returns success trivially if BX equals the block's current size.

Reduction of a block is always possible if there exists a free block immediately above the one being resized. Its base address and size simply need be adjusted to take in the area of memory being shed. Otherwise, a handle must be available from the pool to take up the slack as a new free block. Note the implication. If all handles have been allocated, an XMS client may find it impossible to reduce the size of an EMB.

Enlarging a block is similarly straightforward if a sufficiently large free block sits immediately above. If this is not so, the contents of the block in question must be moved to a new location where enough memory is available to meet the request. Two cases may be identified. If a free block lies immediately below the one being resized, then it may be possible simply to move the existing data downward. Otherwise, a fresh allocation is sought (using the code for HIMEM function 09h or 89h) and assigned to the handle given in DX. The block originally associated with the handle DX is freed (using the code for HIMEM function 0Ah) once its contents have been copied to the new allocation.

Functions 0Fh and 8Fh do *not* test the success or failure of any implicit data transfers (performed by the code for HIMEM function 0Bh). Interrupts will be enabled during such transfers, so the blocks involved are locked temporarily (by direct access to the relevant field in the handle structure) to prevent interference.

HIMEM abhors free blocks of size zero. In the operations described previously, any handles that become associated with free blocks of size zero are returned to the pool.

Versions inspected from 2.50 to 2.77 contain a coding error that affects the two cases where data is moved as part of function 0Fh. The conversion of the source block's size from kilobytes to bytes (ready for the parameter block expected by the code for function 0Bh) is performed with an RCR instruction where an ROR is needed. In consequence, the amount of data transferred during the reallocation is correct only if the original size is a multiple of 64K. Otherwise, the original block's contents will either not be copied at all or some of the last 32K may not be transferred to the new block. The workaround is to use only 64K granularity.

HIMEM versions 3.xx silently correct the previous problem, but function 8Fh and the 80386-specific version of function 0Fh behave incorrectly in the case that requires a fresh allocation to contain an increased size. Registers are restored from the stack in the wrong order, with the result that bp and the high word of edx are corrupted. Instead of releasing the handle that was taken temporarily from the pool to help transfer data between the old and new blocks, the function releases the handle passed in DX.

HIMEM.SYS version 3.09 (distributed with DOS 6) fixes the problem with XMS functions 0Fh/8Fh popping registers in the wrong order on 386 machines. This means that the Reallocate EMB function can finally be relied on to work as advertised. On the other hand, neither this fix, nor the original problem, appears to have been "advertised" at all.

On the Unacceptability of HIMEM 3.xx and the Reallocate EMB Function in General

Readers may have noticed that HIMEM's implementation of the 32-bit functions 88h, 8Eh, and 8Fh is hardly an auspicious beginning for a new interface. The welcome ability of function 88h to return the "highest known physical memory address" is of little use when the function counts the value in kilobytes and does not return it anyway. Similarly, function 8Eh is broken in that it always declares that no handles remain unallocated.

Neither of these problems is devastating. One concerns a new facility and for the other the old form of the function works as advertised. However, consider the implications of the last note describing the implementation of Reallocate EMB in HIMEM.SYS versions 3.xx until its correction for DOS 6. In certain situations, which are very easily arranged once one knows to direct attention to them, functions 0Fh and 8Fh—both the new *and the old*—release the block of memory that the XMS client has asked to be resized. The client program will not know this, because the function will appear to have succeeded. With luck, no other programs will be using extended memory dynamically, and the only consequence will be that the client can no longer perform its intended function, since no more of its XMS calls based on that handle will succeed. A less fortunate event occurs if some other program seeks extended memory later. It may be allocated the handle that the first XMS client believes it owns, so that, before long, one program will probably corrupt the other's extended memory.

Register bp and the high word of edx are corrupted in this same situation. Given the bp register's role in maintaining a stack frame, many programs written in high-level languages will crash as a result of this oversight (which at least has the advantage of making it obvious that something has gone wrong). The register is supposed to be preserved by the XMS server, so few programmers will have seen any need to preserve BP explicitly when using XMS functions.

Let it be stressed that these problems affect not just the new 32-bit variation, but also the ordinary function 0Fh when running on machines with 80386 or better processors. When a program that conforms to the XMS in its use of function 0Fh—and works correctly at close inspection under HIMEM versions 2.xx—is run under HIMEM versions 3.xx, the results may be mysterious failure, memory conflict with other programs, and a system crash. Moreover, *there is no work-around* for new software; the XMS interface does not provide clients with the means to determine in advance whether attempting to resize an EMB will become a reallocate-and-move. The only saving graces are that few real mode programs manage their EMBs dynamically and that HIMEM's implementation of the Reallocate EMB function seems never to have worked correctly anyway.

Even before the large-scale revision of HIMEM.SYS for XMS 3.0, the Reallocate EMB function had a serious defect. Let it be repeated here. If the resizing requires a three-step process (allocate new block, copy data, and release old block), then unless the original block's size is an integral multiple of 64K, either none of the original data will actually be transferred or worse (since it will be far less noticeable), some of the last 32KB will not be copied. Again, software cannot ascertain whether a particular layout of extended memory blocks will run foul of this bug, but at least there is a work-around—to impose a granularity of 64K on all EMBs if the XMS server's function 00 reports major version number 2.

The outstanding question must surely be whether Microsoft has taken appropriate steps to notify developers of these serious faults in the defining implementation of an API that it is largely responsible for promoting and maintaining. Any programmer wanting to use the Reallocate EMB function will have to work around these problems—as described in the previous paragraph for versions 2.xx and by simply not using the Reallocate EMB function if the XMS server's major version is 3 but the total version number is lower than 3.09. Of course, the upper limit in this constraint is known only with hindsight, now that the release of DOS 6 has revealed the problem to have been corrected. It is far from clear how developers were supposed to respond to the situation when it was not known which version of HIMEM would correct the problem.

Microsoft is a company with a wealth of resources, making huge amounts of money (and generating enough publicity about it that the word "gloating" might not be unwarranted) in an industry with little or no regulation, left entirely to its own devices in matters of quality control. It is wholly unsatisfactory that a company in Microsoft's position should oblige others to work around the sloppiness of at least some of its developers, while explicitly disclaiming its responsibility to do better. Add to this the arrogance of declaring that disassembly (perhaps the only source of independent assessment) contravenes the so-called license "agreements" and one has a clear case for imposing moral authority, although unfortunately the world's legal and political institutions do not yet seem up to it.

CHAPTER 13

Using the XMS

In theory, XMS offers programs more power and flexibility in the use of extended memory than do the primitive BIOS extended-memory services provided by int 15h functions 87h and 88h.

Readers may recall that both int 15h functions were put to work in the INT15MEM gadget developed as an example for Part 2. This program began in Chapter 6 as an inert TSR that hooked int 15h function 88h just to take a specified amount of extended memory out of use. In Chapter 7, the program acquired the ability to load as a device driver and was finally allowed to do something with its extended memory block—to behave as a sort of clipboard for temporary storage.

An XMS Clipboard

This chapter develops XMSFILE, a character device driver that uses XMS to emulate a file:

```
device = xmsfile.exe [clustersize] [/Local] [/SMARTDrive]
```

Once installed, the device acts as a clipboard with the device name XMSFILEX. The allocation unit (cluster size) is measured in kilobytes and must be a power of 2. The default is 64KB. Attempts may be made to reclaim extended memory from SMARTDrive if the cluster size is at least 16 and the **/S** switch is given. The device is global by default, that is, an XMS handle is obtained during initialization and although memory is surrendered when the device is not in use, the handle is always kept. The **/L** switch configures the device to seek an XMS handle only when one is needed and to release it when the clipboard is emptied. In this configuration, the device behaves properly under Windows enhanced mode if and only if a Local=XMSFILEX string is placed in the [386Enh] section of SYSTEM.INI and the device is not already in use when Windows is started; each virtual machine accesses its own clipboard, hence its description as local.

Of course, the INT15MEM driver was designed with an upgrade to the XMS in mind. The advantage is the cooperative nature of memory management under the XMS. No memory need be set aside for the clipboard until it is actually needed. Equally important, memory can be surrendered to the system at large when the temporary storage is no longer required.

As written for the int 15h version, one of the driver's component modules—SEQFILE.C—concerns itself solely with emulating sequential file I/O through a character device driver. To this end, it maintains a description of a simulated "file" without regard to the way in which the storage space is found or used; its only concession is that transfer techniques may be limited to moving data in words (as are both the relevant int 15h and XMS functions). Accordingly, that module may be adopted for the XMS version without change. Material dependent on the use of int 15h functions for allocation and access is cleanly separated into other modules—MAIN.C and TRANSFER.C—which now simply need be replaced with XMS versions.

```
/* transfer.c -- XMS version */

/*  This file (suitable for use as a resident module under the Keep C scheme
    for resident programming) accompanies the character device driver
    emulation of a sequential file in extended memory. It contains the
    material that depends on use of the XMS to manage extended memory blocks. */

#include     <driver.h>
#include     <keepc.h>
#include     <standard.h>

#include     "xmsfunc.h"

/*  Variables describing the file state - declared in SEQFILE.C  */

extern DWORD dwFilePtr;
extern DWORD dwFileEnd;

/*  The XMS calling address - recorded at initialization. */

void (_far *lpXMS) (void) = NULL;

/*  Variables describing the Extended Memory Block (EMB) obtained from the
    XMS server      */

static DWORD dwEMBSizeInBytes = 0;
static DWORD dwEMBSizeInK = 0;
static WORD wEMBHandle = 0;

/*  A structure for the XMS function that moves data to and from extended
    memory  */

static struct _XMS_MOVEBLOCK {
    DWORD length;
    WORD source_handle;
    DWORD source_offset;
    WORD dest_handle;
    DWORD dest_offset;
} moveblock = {0};

/*  A parameter describing the allocation unit as a binary logarithm of the
    size in K—this is recorded at initialization. */

WORD wLogClusterSize = 16;            //  default cluster size is 64K

/*  A Boolean variable which is FALSE by default but may be set to TRUE
    (during initialization) if extended memory should be sought from the
    SMARTDrive disk cache when desparate. */

BOOL bUseSMARTDrive = FALSE;

/*  The total amount (measured in K) by which the cache has been reduced—
    maintained so that an attempt can be made to restore the cache when the
    device is closed. */

static WORD wCacheDeficit = 0;

/*  External functions for manipulating the SMARTDrive cache  */

extern BOOL DecreaseCache (DWORD);
extern BOOL IncreaseCache (DWORD);

/*  A flag recorded during initialization which determines whether the XMS
    handle should be released whenever the device is not in active use      */

BOOL bLocal = FALSE;

/*  ========================================================================  */

/*  SEQFILE.C requires the following function in order to know how to
    transfer data. The command argument may be D_READ or D_WRITE to
    indicate whether data is being read from or written to extended
    memory, respectively. */

BOOL MoveBlock (BYTE command, BYTE _far *lpBuffer, DWORD xmptr, WORD count)
{
    /*   The XMS interface uses a natural description of the transfer
         regions, so the function need simply record its arguments in the
         parameter block. The EMB is accessed through a handle; the linear
         address of the block need not be known. The buffer in real-mode
         addressable memory is described by a null handle. */
    moveblock.length = count;
    if (command == D_READ) {
          moveblock.source_handle = wEMBHandle;
          moveblock.source_offset = xmptr;
          moveblock.dest_handle = 0;
          moveblock.dest_offset = (DWORD) lpBuffer;
    }
    else if (command == D_WRITE) {
          moveblock.source_handle = 0;
          moveblock.source_offset = (DWORD) lpBuffer;
          moveblock.dest_handle = wEMBHandle;
          moveblock.dest_offset = xmptr;
    }
    else return (FALSE);

    /*    Call the XMS server, returning a Boolean status (ignoring any XMS
          error code). */

    _asm {
               mov      si,OFFSET moveblock      ; the compiler preserves si
               mov      ah,0Bh
               call     lpXMS
    }
}

/*  The function AdjustRequest is called by SEQFILE to determine how much of
    a read or write request can actually be met. Its return value is
    necessarily less than or equal to the DWORD argument. If asked to write
    more data than the EMB can accommodate, this XMS version of the function
    will try to expand the current allocation.

    Note that this implementation is fairly simplistic and expects
    that the caller will be interested mainly in whether or not the whole
    request can be accommodated. Thus, if meeting the request demands
    expansion of the EMB, which turns out to be impossible, the present
    implementation does not try expanding the EMB gradually but simply trims
    the request to the amount of unused space currently in the EMB. Against
    this, the function tries very hard to arrange the EMB's expansion, even
    to the extent of stealing memory from the SMARTDRV.SYS disk cache. */

DWORD AdjustRequest (BYTE command, DWORD numwanted)
{
    DWORD avail, extrabytes, extraclusters, extrakb, newsizeinkb;
    WORD cache_tried;
```

```
/*   If the driver has no EMB to read from or write to, stop the
     request. */

if (wEMBHandle == 0) return (0);

/*   Compare the request to the amount of available space. Read
     operations are limited to the area of the file to which data has
     already been written. */

avail = (command == D_READ ? dwFileEnd : dwEMBSizeInBytes) - dwFilePtr;
if (numwanted > avail) {

     /* Simply reduce the size of any request to read past the end of
        the file. */

     if (command == D_READ) return (numwanted = avail);

     /* The XMS offers the opportunity of resizing the EMB to
        accommodate an attempt to write more data than will fit.
        Control the file's growth by allocating extra memory in
        multiples of some cluster size fixed during initialization (to
        be some power of two when measured in kilobytes). A larger
        cluster size will mean fewer calls to the XMS server but may
        also mean using more extended memory than necessary. */

     extraclusters = shr (numwanted - avail - 1, wLogClusterSize) + 1;
     extrabytes = shl (extraclusters, wLogClusterSize);
     extrakb = shr (extrabytes, 10);

     /* Only with version 3.0 of the XMS interface is it possible to
        accommodate block sizes of 64MB or more. This demonstration
        does not use the 32-bit XMS function 8Fh introduced to deal
        with such large EMBs, partly because the arrangements required
        under Microsoft C 6.0 to use 32-bit registers for the XMS call
        are prohibitive, but mostly because the extant implementation
        (HIMEM.SYS version 3.07 distributed with Windows 3.1) is
        seriously flawed. */

     newsizeinkb = dwEMBSizeInK + extrakb;
     if (HIGHWORD (newsizeinkb) != 0) return (numwanted = avail);

     /* If the first attempt to resize the EMB fails, it may be
        worth seeking the extra space from the SMARTDrive disk
        cache and try again. */
        cache_tried = 0;
        for (;;) {
            if (ResizeEMB (wEMBHandle, (WORD) newsizeinkb)) {
                  dwEMBSizeInBytes += extrabytes;
                  dwEMBSizeInK += extrakb;
                  break;
            }
            if (bUseSMARTDrive) {
                  cache_tried ++;
                  if (cache_tried == 1) {

                      /*    It is preferable to ask the cache to surrender only
                            the marginal requirement, but this will probably
                            only work if the disk cache is directly below the
                            XMSFILE EMB, or trivially if XMSFILE has no EMB. */

                      if (DecreaseCache (extrakb)) {
                            wCacheDeficit += extrakb;
                            continue;
                      }
                  }
                  else if (cache_tried == 2) {

                      /*    If it turns out that the block still cannot be
                            expanded, then try shrinking the cache so far that
                            the memory released will be enough to contain an
                            enlarged XMSFILE block. */

                      if (DecreaseCache (dwEMBSizeInK)) {
                            wCacheDeficit += dwEMBSizeInK;
                            continue;
                      }
                  }
            }

            /*    If all this cache manipulation fails to help, tailor the
                  request to fit the available space. */

            numwanted = avail;
            break;
        }
    }
    return (numwanted);
}
```

```
/*  Device-driver functions supplied in SEQFILE.C for the open and close
    commands ensure that the file pointer is reset when the device is opened
    and that the file is truncated at the last position read or written when
    the device is closed. Open and close features specific to the method of
    extended memory management are provided by the following functions. */

WORD Open (void)
{
    DWORD temp;

    if (wEMBHandle == 0) {
          dwFileEnd = 0;
          dwEMBSizeInBytes = 0;
          dwEMBSizeInK = 0;
          temp = AllocateEMB (0);
          if (NOT (BOOL) temp) return (D_ERROR | D_DONE | D_GENERAL_FAILURE);
          wEMBHandle = HIGHWORD (temp);
    }
    return (D_DONE);
}

WORD _far FarOpen (void)                  // provided so that the device may
{                                         // be opened during initialization
    return (Open ());                     // if configured to act globally
}

WORD Close (void)
{
    DWORD newclusters, newk, newsize, returntocache;

    if (wEMBHandle == 0) return (D_DONE | D_ERROR | D_GENERAL_FAILURE);

    /*    On closure, trim the EMB of any extended memory that is not being
          used to hold data. */

    newclusters = dwFileEnd ? shr (dwFileEnd - 1, wLogClusterSize) + 1 : 0;
    newsize = newclusters ? shl (newclusters, wLogClusterSize) : 0;
    newk = newsize ? shr (newsize, 10) : 0;

    if (newk < dwEMBSizeInK OR (newk == 0 AND dwEMBSizeInK == 0)) {

          returntocache = 0;

          /* For a local device, the EMB handle can be released altogether
             (and the wEMBHandle variable cleared) whenever the size returns
```

```
                to zero. */

            if (bLocal AND newk == 0 AND ReleaseEMB (wEMBHandle)) {
                wEMBHandle = 0;
                returntocache = dwEMBSizeInK;
            }
            else if (ResizeEMB (wEMBHandle, (WORD) newk)) {
                returntocache = dwEMBSizeInK - newk;
                dwEMBSizeInK = newk;
                dwEMBSizeInBytes = newsize;
            }
            else return (D_DONE | D_ERROR | D_GENERAL_FAILURE);

            /*  If the SMARTDrive cache has been shrunk to obtain more memory
                for the file emulation, attempt to return as much of this
                as possible out of the amount just released. */

            if (wCacheDeficit AND returntocache) {
                if (returntocache < wCacheDeficit) returntocache = wCacheDeficit;
                if (IncreaseCache (returntocache)) wCacheDeficit -= returntocache;
            }
        }
        return (D_DONE);
}

WORD Destroy (void)
{
        /*  This function is needed only for an intelligent local device. It
            would be used, on notification from the switcher, that a task is being
            destroyed. */

        Close ();           //   destruction may preempt explicit closure
        _asm {
                xor     dx,dx
                xchg    dx,wEMBHandle
                mov     ah,0Ah
                call    lpXMS
        }
        return (D_DONE);
}
```

Task Switching

This simple implementation of TRANSFER.C offers no explicit support for task-switching but must nonetheless be prepared, at least defensively, for the possibility of operating in a task-switching environment. Take stock of this, for it is faintly perverse. In its default configuration, XMSFILE does not expect to use any task-switcher functionality, but despite this, its coding is constrained by task-switching considerations.

In particular, although it would be economical for XMSFILE to defer its XMS allocation until some program actually opens the clipboard and to release the XMS handle whenever the clipboard is emptied, this practice would be unsafe. Instead, the device driver's EMB must be secured before the first task switch (hence the unnatural provision in TRANSFER.C for a far call to the `Open` function) and must never be released. The only real alternative is to design XMSFILE to communicate with the task switcher in various ways, depending on how it would like the clipboard to appear to distinct tasks.

Similar considerations apply, incidentally, to TSRs that allocate conventional memory, say for file I/O, during a pop up stage. If the TSR is started before the task switcher, all sorts of confusion can result when the TSR pops up in one task and is then switched into another. The trouble stems from the way that new conventional-memory allocations are taken from an area of memory that the switcher localizes task by task. Switching to a different task is like pulling the rug out from underneath the pop-up TSR if it doesn't realize that the memory it just allocated is no longer present, having been swapped out to disk or mapped away from the addresses where the TSR expects to access it. A rough solution, probably not implemented by many task switchers, is to note the current PSP when a memory allocation is sought within a task; if the PSP is in global memory and the process appears to be unaware of the switcher's existence (for instance, it has neither identified instance data nor hooked a notification chain), then the switcher should arrange for the memory allocation either to fail or to be global.

The main point for modern programming is that, however much it would be pleasing to think that a task switcher would be designed to interfere as little as possible with programs that are unaware of the switcher's presence, and however much this may be achievable for programs that run wholly within the switcher (accessing no resources that are not wholly within the switcher's control), there will always be incompatibilities with software such as TSRs that are loaded before the switcher and that continue to use system resources while other programs come and go. The other side of this coin, of course, is whether the extant task switchers make sufficient effort to resolve these intrinsic incompatibilities or merely use their existence as a cover for lazy implementations.

Localized Extended Memory

The particular concern is with a task switcher's treatment of extended memory and the possible existence of intrinsic incompatibilities that must be catered for in XMSFILE's design. Just as each task has the use of all free conventional memory while it is active, so too may a task switcher aim to give each of its clients access to all the available extended memory, one at a time. The difficulty for a task switcher is that the XMS treats extended memory allocation very much as a private affair between the XMS server and each client. There is no facility by which the task switcher can discover what extended memory a task is using except (as studied in the next chapter) by hooking itself into the XMS interface and playing the role of XMS server to each task. A handle obtained inside a task-switching session may be just a confection, a number invented by the switcher to describe a subregion of an EMB that the switcher has obtained from the genuine XMS server.

If programs running under a task switcher access the clipboard, then XMSFILE may find itself with one foot on each side of the boundary between two different XMS environments. As already noted, the default configuration obtains an XMS handle during the device driver's initialization and retains the handle permanently. This way, XMSFILE will only ever use a global resource. If a switcher is started, it will have to respect the prior allocation of an XMS resource; all requests for access to the XMS clipboard will have to be passed to the original XMS server rather than be managed by the switcher.

Suppose instead that XMSFILE were written to delay its allocation of an XMS handle until software actually attempts to access the extended memory clipboard. This is an attractive design goal since it means that no XMS resources, not even a handle for a zero-byte EMB, are used until they really are needed. Observe that TRANSFER.C sets out to accomodate this idea by allocating the EMB when the driver receives a request to open the device before an XMS handle has been obtained. Suppose also that the software that first asks for access to the clipboard happens to be running under the control of a task switcher. In this situation, the handle that XMSFILE stores in the appropriate internal variable may be just an artifact of the switcher's bookkeeping. It may be meaningful only in the context of this particular task, but because XMSFILE is in global memory, the clipboard will always be accessed through this same XMS handle no matter what the current task and even after the switcher has terminated.

Task Separation

If the task switcher guarantees that two tasks cannot be given the same XMS handle, then a task switch will cause no real concern. The XMS handle stored by XMSFILE will be unusable except by the task that was running when the handle was allocated. The most likely outcome of attempting to access the clipboard from a second task is a critical error as the XMSFILEX device finds that its XMS handle suddenly seems to be invalid.

On the other hand, a task switcher may distribute the same XMS handle to any number of tasks that ask for extended memory. The thinking here is that the switcher always knows which task has issued an XMS request and is therefore always able to separate two tasks' use of the one handle. The effect is that the switcher defines a complex handle composed of the original 16-bit XMS handle together with whatever ID (such as a PSP) the switcher uses to distinguish its tasks. This may suit the switcher's purposes and is indeed immaterial to software running under the switcher, but it is very inconsiderate for software that happens to be in global memory (loaded *before* the switcher) and which believes, quite reasonably, that an XMS handle is sufficient information to specify an extended memory block.

In the first case, the result is not very satisfactory (if the clipboard is not used until the switcher has started, then it will be available to just the one task), but at least it is not unsafe. With the second type of switcher, memory corruption is just around the corner. Imagine that an XMS handle for the XMS clipboard is obtained while task A runs. Task B then allocates extended memory for its own purposes and happens to be given the same XMS handle that was given to task A. If task B subsequently attempts to write to the clipboard, then the XMSFILEX device, not realizing that the handle it has saved (in global memory) was intended only for task A, will corrupt the other extended memory block belonging to task B.

This is a disastrous scenario, so it is tempting to reject the second type of switcher behavior as incompatible with XMS. Were the second type to exist in the real world, all resident XMS clients would have to

- load as device drivers and allocate XMS resources during their initialization (on the reasonable assumption that this precedes the execution of the task switcher);
- implement code to detect the switcher's initialization and then to allocate XMS resources before the switcher starts any tasks; or
- cooperate explicitly with the switcher's localization of XMS resources among tasks.

There is every hint in the XMS documentation that programs should use XMS resources economically. However, except for device drivers, programs that have no knowledge of the switcher's interfaces, but which sit in memory waiting to allocate XMS resources only when they are actually needed, would be broken by the second type of switcher.

Readers are probably cynical enough by now to have already guessed that this second type of switcher not only exists in the real world but has as its prime example a successful Microsoft product: Windows enhanced mode. For an exercise, start Windows enhanced mode and open two DOS Prompts, both configured (through the relevant PIF setting) to supply extended memory. In the first DOS session, run DEBUG, then assemble and execute the instructions

```
mov       ax,4310
int       2F
```

to locate the XMS calling point. Suppose this to be 0266:00CE (as it happens to be on the author's machine running the HIMEM.SYS 3.09 distributed with DOS 6.0). Now assemble instructions to obtain a zero-byte EMB:

```
xor       dx,dx
mov       ah,09
call      0266:00CE
```

Assuming this returns successfully, switch to the other DOS Prompt in order to also seek a zero-byte EMB in the second session. In both cases, the debugger's listing of registers will show the XMS handle in dx. It will be the *same* value for both virtual machines.

Note the implication here: Even though the XMS is defined primarily to give real-mode DOS programs a well-defined, convenient way to use extended memory, resident DOS programs intending to use the XMS *must* accommodate the special behavior of Windows enhanced mode. Substitute the name of any other program for Windows enhanced mode and the previous sentence would meet many developers' notions of incompatibility.

When one program hooks into an existing system, surely it is the responsibility of the new program to anticipate the reasonable behavior of software already present in the existing system and to strive for compatibility. Of course, the benefits of Windows enhanced mode are arguably very great, so there is some call for compromise, but even so, one may expect known incompatibilities to be well documented in places that will be regarded as accessible by programmers working in the context of the original system.

Thus, it is one thing for Microsoft to inform programmers that

> a global xMS user who ALLOCATES an xMS handle, UMB or the HMA while WIN386 is up will very likely malfunction because these objects will be LOCAL to the VM they are allocated in. For this reason Global xMS users need to ALLOCATE ALL their xMS objects with the correct size at initialization.

but quite another that this appears in a document (the Windows/386 Paging Import Specification) that looks for all the world like a specialist treatise intended only for the writers of EMS emulators like EMM386.EXE, which need to arrange an orderly transfer from one protected mode environment to the Windows environment and back again. It is far from clear that many DOS programmers need to or will want to understand the intricacies of the V86MMGR virtual device and the highly specialized (and substantial) interface covered in that document. Yet this is the only place they will learn of the constraints that Windows puts on software present *before* Windows.

Global XMS Allocation from Within a Task Switcher

One thing DOS programmers will find in the XMS documentation is that

> Allocation of a zero length extended memory buffer is allowed. Programs which hook XMS drivers may need to reserve a handle for private use via this method. Programs which hook an XMS driver should pass all requests for zero length EMBs to the next driver in the chain.

Pay particular attention to the last sentence, which means that if a task switcher conforms to this advice when hooking XMS services for its clients, then although XMS allocations made while the switcher is running will ordinarily be *local* to the task seeking the service, a program under the control of the task switcher may obtain a *global* XMS handle by asking for an EMB of size zero.

Now, XMSFILE's use of XMS services has been written to match the device driver interface as efficiently as possible. No extended memory is actually needed until the device driver Write function is called and none is sought until then. It would be possible, and indeed very economical, to delay the allocation of an XMS handle until a non-empty EMB is required, but the XMSFILE driver implements a two-step allocation to correspond to the typical pattern of device access. Thus, when the device is opened, this is taken as a signal to obtain the basic resource: an XMS handle not yet associated with any memory. Then, when the device is written to, the memory is sought by calling the XMS function that resizes EMBs.

An immediate corollary of this behavior is that in the presence of a task switcher conforming to the piece of XMS documentation already noted, XMSFILE may escape the usual requirement of having to allocate its XMS resources during initialization. It may indeed wait until the clipboard is opened before it reserves an XMS handle and it may release the handle whenever the clipboard is closed while empty.

This behavior is supported by TRANSFER.C above if the `bLocal` variable is TRUE and the `Open` function is not called during the device driver's initialization. These two properties are arranged in turn by the MAIN.C module below when it sees the **/local** switch on the device driver's command line. Note, however, that in the presence of a switcher that translates requests for empty EMBs into global XMS allocations, the practical effect of **/local** is still a global clipboard, but one which makes slightly more elegant, economical use of XMS resources.

Now, it might be thought that all switchers that hook into the XMS would follow the documentation. However, the context is a section giving "Implementation Notes for DOS XMS Drivers," and this appears to be sufficient licence to excuse Windows enhanced mode from conforming to the rules laid down for other software. There is perhaps nothing wrong with this from the Windows perspective, but it creates a problem for DOS programmers because those who write XMS client software may easily interpret the XMS documentation as assurance that global allocations can be arranged by asking for a zero-length EMB. Perhaps that is too superficial an interpretation, but in any case, under Windows enhanced mode (but not under the MS-DOS Shell's task switcher or the closely related task switcher for Windows 3.1 standard mode, which both conform), the **/local** variation starts to show where its name comes from. XMS resources for the clipboard will not be reserved during the device driver's initialization, so if the first access to the XMSFILE clipboard occurs in a virtual machine, the handle will be local. For all the reasons discussed in the preceding pages, XMSFILE must cooperate with Windows to complete the per-VM localization of the clipboard.

A Local XMS Clipboard Under Windows Enhanced Mode

The elegant solution is for XMSFILE to wait for Windows to issue its int 2Fh function 1605h initialization broadcast. If the clipboard is not-empty when Windows starts, then XMSFILE must configure itself for global usage, meaning in particular that it must make sure never to release its global XMS handle. On the other hand, if the clipboard is empty when Windows starts, then localization is easily arranged just by nominating as instance data those few variables that define the state of the clipboard. When suspending a VM, Windows will save these variables for restoration when that VM resumes. In effect, each VM receives its own XMS clipboard even though this local clipboard is managed by code in global memory.

TRANSFER.C accommodates linkage with an extra module offering explicit support for a task-switcher interface. Extension in this direction is left as an exercise for the reader, but the following does describe a poor man's localization under Windows enhanced mode.

Whereas the elegant solution arranges for instancing just the necessary data and configures the device globally or locally depending on the status when Windows starts, there is a rough and ready provision that demands no extra coding but which produces a local clipboard, admittedly without the fallback to global operation if the clipboard is already in use. Adding the line **Local=XMSFILEX** to the **[386Enh]** section of SYSTEM.INI directs the DOSMGR virtual device to instance all memory belonging to the XMSFILEX device. Starting with Windows 3.1, there is an equivalent **localtsrs=** entry in the **[NonWindowsApp]** section of SYSTEM.INI for programs (as opposed to device drivers) in global memory. In many cases, this simple ruse suffices until futher development of the program or driver leads to proper support for localization.

In the case of XMSFILE, there are some global variables that record changes made to the SMARTDRV.SYS disk cache and which should not be instanced, but in practice it is rarely possible to resize the cache while Windows is running (and in any case it is probably not a good idea), so it does not matter too much if the relevant variables are instanced when technically they shouldn't be. The other main restriction arising from use of the poor man's instancing technique is that the XMSFILE clipboard will misbehave unless it is empty when Windows starts.

Initialization

The XMSFILE initialization file, MAIN.C, exists simply to set operating conditions for the storage space. It reads and validates parameters from the device driver command line and arranges the desired configuration of the TRANSFER.C module.

```
/* main.c - initialization file for XMSFILE */

#include    <dos.h>
#include    <stdarg.h>
#include    <stdio.h>
#include    <stdlib.h>
#include    <string.h>

#include    <driver.h>
#include    <keepc.h>
```

```
#include    <standard.h>

static CHAR description [] = "\n\
XMSFILE is a character device driver that uses XMS to emulate a file.\n\
\n\
    device = xmsfile.exe [clustersize] [/Local] [/SMARTDrive]\n\
\n\
Once installed, the device acts as a clipboard - with the name XMSFILEX.\n\
\n\
The allocation unit (clustersize) is measured in kilobytes\n\
and must be a power of 2. The default is 64K.\n\
\n\
Attempts may be made to reclaim extended memory from SMARTDrive\n\
if the cluster size is at least 16 and the /s switch is given.\n\
\n\
The device is global by default, that is, an XMS handle is obtained during\n\
initialization and although memory is surrendered when the device is not in\n\
use, the handle is always kept. The /l switch configures the device to\n\
seek an XMS handle only when one is needed and to release it when the\n\
clipboard is emptied. In this configuration, the device behaves properly\n\
under Windows enhanced mode if and only if a Local=XMSFILEX string is\n\
placed in the [386Enh] section of SYSTEM.INI and the device is not already\n\
in use when Windows is started;  each virtual machine accesses its own\n\
clipboard, hence its description as local.\n";

/*  A flag defined by the CRTDRVR library to enable programs to determine
    whether or not they are being run to initialize a device driver or as
    a normal DOS executable  */

extern BOOL _RESIDENT bLoadedAsDriver;

/*  A function to find the address of the XMS driver's calling point and a
    resident variable in which to record this address  */

extern void (_far *GetXMSAddress (void)) (void);
extern void (_far * _RESIDENT lpXMS) (void);

/*  A parameter describing the XMSFILE's allocation unit  */

extern WORD _RESIDENT wLogClusterSize;

/*  A resident function to obtain an initial memory allocation by opening
    the device */

extern WORD _far FarOpen (void);
```

```
/*  A flag denoting that extended memory should be recovered from SMARTDrive
    should no memory be available by normal means—and a function to
    prepare this facility  */

extern BOOL _RESIDENT bUseSMARTDrive;
extern BOOL InitSMARTDrive (WORD);

/*  A flag in resident memory to record that the driver should attempt to
    behave locally to each Windows virtual machine—plus a warning message  */

extern BOOL _RESIDENT bLocal;

static char localwarning [] = "\n\
For correct behavior under Windows enhanced mode, the device requires\n\
the string Local=XMSFILEX in the [386Enh] section of SYSTEM.INI\n\
and must not be in use when Windows is started\n";

/*  Miscellany  */

static WORD log (WORD);
void PutError (char *, ...);
void PutWarning (char *, ...);

/*  ========================================================================  */

int main (int argc, char **argv)
{
    register CHAR *ptr;
    register WORD clustersizeinkb;
    BOOL clustersizedone = FALSE, smartdrivedone = FALSE;
    WORD temp;
    struct find_t fileinfo;

    /*    If XMSFILE.EXE is executed as a DOS program, rather than as a device
          driver, display a message describing its function and syntax. */

    if (NOT bLoadedAsDriver) {
          printf (description);
          return (0);
    }

    /*    Avoid multiple loading: abort the installation if an XMSFILEX
          device or file exists already. */

    #define _A_GENERAL       (_A_HIDDEN | _A_SYSTEM | _A_SUBDIR)
```

```
if (_dos_findfirst ("XMSFILEX", _A_GENERAL, &fileinfo) == 0) {
      PutError ("XMSFILEX already present");
      return (-1);
}

/*    Parse the command line. */

while (++ argv, -- argc) {
      ptr = *argv;
      if (*ptr == '/') {
          ptr ++;

          /*    The switches understood by the program need not be
                entered in full—any initial substring is acceptable. */

          if (strnicmp ("SMARTDrive", ptr, strlen (ptr)) == 0) {
                smartdrivedone = TRUE;
                continue;
          }
          if (strnicmp ("Local", ptr, strlen (ptr)) == 0) {
                bLocal = TRUE;
                continue;
          }
          ptr --;
          PutError ("invalid switch %s", ptr);
          return (-1);
      }
      else {
          if (clustersizedone) {
                PutError ("too many parameters");
                return (-1);
          }

          /*  A numerical parameter must be supplied—XMSFILE's memory
                allocation will be adjusted in multiples of this many K,
                defaulting to 64. */

          clustersizeinkb = atoi (ptr);
          if (clustersizeinkb == 0) {
                PutError ("expected cluster size, got %s", ptr);
                return (-1);
          }
          temp = log (clustersizeinkb);
          if (temp == -1) {
                PutError ("cluster size must be power of 2");
                return (-1);
            }
            wLogClusterSize = temp + 10;
            clustersizedone = TRUE;
        }
    }
    if (NOT clustersizedone) {     // the default cluster size must be 64K
        clustersizeinkb = 64;           // to ensure proper behavior of XMS
        wLogClusterSize = 16;            // function 0Fh in versions 2.xx of
    }                                     // HIMEM.SYS.

    /*  If the option to steal memory from SMARTDrive (when necessary) has
        been exercised, check its feasibility. It is the responsibility of
        the SMARTDrive-specific testing function to issue any warning
        message and disable the option, if conditions are not suitable. */

    if (smartdrivedone) bUseSMARTDrive = InitSMARTDrive (clustersizeinkb);

    /*  Check that an XMS server is installed and, if so, record its calling
        address. */

    lpXMS = GetXMSAddress ();
    if (lpXMS == NULL) {
        PutError ("no XMS server installed");
        return (-1);
    }

    /*  Allocate an Extended Memory Block. This step would not be needed in
        a single-tasking environment or if task-switching were fully
        supported (since in both cases, the driver could simply wait for an
        attempt to open the device). */

    if (NOT bLocal) {
        if (FarOpen () & D_ERROR) {
            PutError ("Extended memory handle unavailable");
            return (-1);
        }
    }

    /*  Indicate successful installation, then tidy up. */

    printf ("\nXMSFILEX installed. Cluster size = %dK.\n", clustersizeinkb);
    if (bLocal) printf (localwarning);

    CloseAllFiles ();
    FreeEnvironment ();
    ExitKeepC (0, 0);
}

/*  The logarithm of a power of 2 may be computed easily enough by shifting
    bits to count the number of trailing zeros. The function returns the
    impossible value 0xFFFF if given an argument that is not a power of 2. */

static WORD log (register WORD x)
{
    register WORD n = 0;
    if (x) {
         while ((x & 1) == 0) {
              x >>= 1;
              n ++;
         }
         if (x == 1) return (n);
    }
    return (0xFFFF);
}

void PutError (char *str, ...)
{
    va_list va_start (argptr, str);
    printf ("\nXMSFILE intialization error:  ");
    vprintf (str, argptr);
    printf ("\n");
    va_end (argptr);
}

void PutWarning (char *str, ...)
{
    va_list va_start (argptr, str);
    printf ("\nXMSFILE warning:  ");
    vprintf (str, argptr);
    printf ("\n");
    va_end (argptr);
}
```

There is still some way to go before the program will be presented in a final version, for the opportunity is taken later to incorporate interaction with the SMARTDrive disk cache. It may help to consider (now) some practical effects of configuring the device for local behavior under Windows enhanced mode. Place

```
device xmsfile.exe
```

in CONFIG.SYS with an appropriate path and reboot. Run Windows and start two DOS boxes. In the first, enter the command

```
type system.ini >xmsfilex
```

and in the second, copy WIN.INI instead. In this case, the XMSFILEX device obtained its EMB during startup and, although the two commands are issued in separate virtual machines, they access the same storage. This may be seen by returning to the first DOS box and retrieving the contents of the XMS clipboard

```
type xmsfilex
```

only to find WIN.INI scroll over the screen. With this configuration the device is global and may be used to transfer text from one virtual machine to another or between virtual machines and the DOS session outside Windows.

Now edit CONFIG.SYS to give XMSFILE.EXE its **/local** switch, reboot, and, before using the clipboard or running Windows, edit SYSTEM.INI to contain a **Local=XMSFILEX** line in the **[386Enh]** section. Start Windows enhanced mode and repeat the experiment with the two DOS boxes. This time, returning to the first machine finds the XMS clipboard containing a copy of SYSTEM.INI, despite the intervening transfer of WIN.INI by the second DOS box. The device has been localized; in a very real sense, each virtual machine has its own copy of the XMSFILEX device which acts entirely independently of its counterparts in other DOS boxes.

Memory from SMARTDrive

Many XMS clients have fixed requirements (as has a RAM disk, for example), but whenever possible, programs should remember that the system as a whole is better served by cooperative use of valuable resources and should therefore release memory for which other programs may have greater need. In the case of XMSFILE, the situation is clearly marked. The simulated "file" in extended memory is truncated by closure, and, since memory beyond that point cannot be used again until the device is reopened, it ought to be returned to the XMS server. Other cases might not offer clear a signal. If the possibility exists that some of a program's extended memory allocation could be put to better use elsewhere, then thought should be given to providing an interface through which other programs may express their need for more memory.

A good example of this cooperative attitude to extended-memory usage is offered by the SMARTDRV.SYS disk cache, supplied as standard with DOS 5 (but replaced in Windows 3.1 and future DOS releases by a new SMARTDRV.EXE, with which it has little in common). This program monitors BIOS-level disk activity, building a collection of tracks in memory to reduce the overhead incurred by repeated access to the same portions of a disk. By default, the cache is built in extended memory. At installation, two parameters may be given on the device-driver command line—one is the amount of memory to obtain for the cache and the other (said to be useful only when Windows is running) is the minimum to which this allocation may be reduced. Thus, SMARTDrive is prepared to cede some or all of its cache to any program that declares a better use for the memory—as long as the request is delivered in terms that SMARTDrive understands. In practice, this means that SMARTDrive surrenders memory only to Windows, but the necessary interface is documented in a SMARTDRV.INC header file distributed with the Windows DDK, so other programs may resize the cache to gain extended memory for their own use, just like Windows. To illustrate this XMSFILE has been given a token ability to ask SMARTDrive to return a little memory to the XMS pool when at first there is not enough to meet XMSFILE's plans for expansion.

Incidentally, the study of SMARTDRV.SYS is conducted not just to give XMSFILE a little extra power, but because SMARTDRV.SYS is one of the few readily available programs that actually manage extended memory dynamically from real mode. As such, much of the material (and program code) developed here will be built into a tool for the investigation of typical extended memory environments after completion of the big XMS mapping utility in Chapter 15.

It will be seen shortly that the SMARTDRV.SYS interface is no small concern. Readers may wonder why no SMARTDrive configuration program was ever distributed (at least not actively). On the other hand, readers may also remember the comments at the end of Chapter 12 that XMS function 0Fh for resizing extended memory blocks has never worked correctly, at least in the extant implementation provided by HIMEM.SYS. The new SMARTDRV.EXE is declared to be more flexible than its predecessor, but while it offers statistics, drive-by-drive configuration, and so forth, it does not provide the ability to resize the cache (except to Windows). Cynics may wonder if these several points are somehow connected.

The SMARTDRV.SYS Interface

Communication with SMARTDRV.SYS (but not the newer SMARTDRV.EXE) relies exclusively on the IOCTL device-driver functions. To initiate an exchange, the device driver should be opened under the name "SMARTAAR". An IOCTL Read (int 21h function 4402h) returns information about SMARTDrive's performance and configuration, as detailed in Table 13-1. At least 28h bytes of control data must be requested, or the IOCTL Read function will return a transfer count of zero. The full structure is 2Ch bytes long and it is recommended that 2Ch bytes be sought.

The configuration data may establish that the cache is using extended memory and reveal a difference between the cache's current size and the minimum size set during SMARTDrive's initialization. In this case, software may want to reduce the cache size in order to gain more extended memory for its own purposes.

Table 13-1: Data Structure for SMARTDrive IOCTL Read

The transfer count will be truncated to zero, 28h, or 2Ch, as applicable.

OFFSET	SIZE	DESCRIPTION
00h	byte	unused, except that it may be changed by IOCTL Write function 04h subfunctions 00h and 01h
01h	byte	unused, except that it may be changed by IOCTL Write function 04h subfunctions 02h and 03h
02h	byte	00h if the cache has been deactivated; 01h if active
03h	byte	01h if the cache is in extended memory; 02h if in expanded memory
04h	word	number of timer ticks between flushes—default interval is 1 minute but current versions do not actually install the int 1Ch handler
06h	byte	00h normally; 01h if the cache contains tracks marked nondiscardable
07h	byte	00h normally; 01h to ensure that the cache be flushed when int 19h is received
08h	byte	unused, except that it may be changed by IOCTL Write function 0Ah subfunctions 00h and 01h
09h	byte	00h if double-buffering is disabled; 01h if double-buffering is forced; 02h if the need for double-buffering is determined dynamically
0Ah	dword	address of the handler which is immediately below SMARTDRV in the int 13h chain
0Eh	word	SMARTDrive version number (minor version in low byte)
10h	2 bytes	unused
12h	3 words	number of sectors attempted to be read, number found in the cache, and number found in the intermediate buffer. Only ratios should be regarded as meaningful, since all three values are halved whenever one of them is about to overflow.
18h	2 bytes	statistical information in the form of percentage ratios for cache hits and buffer hits respectively. These are maintained over the whole of the session and are not cleared by resetting the cache.
1Ah	word	number of tracks the cache can hold
1Ch	word	number of valid tracks in the cache

Table 13-1: Data Structure for SMARTDrive IOCTL Read

OFFSET	SIZE	DESCRIPTION
1Eh	word	number of nondiscardable tracks in the cache
20h	word	number of dirty tracks in the cache
22h	word	current cache size (in multiples of 16K)
24h	word	maximum cache size (in multiples of 16K)
26h	word	minimum cache size (in multiples of 16K)
28h	dword	address of a flag for locking the cache (used by the PAGESWAP and PAGEFILE virtual devices in Windows 3.0 and 3.1, respectively, to avoid cache disruption during int 13h access to the permanent swap file)—00h by default, but a nonzero value locks the cache's current contents (so that a new track may be entered in the cache only if an empty track buffer exists, not by displacing an existing track, whether discardable or not)

Cache manipulation is supported by an IOCTL Write (int 21h function 4403h) interface that stretches to several subfunctions by interpreting the first byte of control data as a command number. Full details are presented in Table 13-2. Observe that separate commands are provided for increasing and decreasing the cache, rather than a single command to specify an absolute value for the new size.

Table 13-2: SMARTDrive IOCTL Write Commands

The data packet contains a command number as its first byte. For some commands, extra data (a byte, word, or dword immediately after the command number) may be required. In all cases, an insufficiency will be indicated by returning a transfer count of zero bytes.

COMMAND	DATA	DESCRIPTION
00h	-	flush all dirty tracks to disk
01h	-	flush and reset
02h	-	deactivate cache (flush, reset, and disable)
03h	-	activate cache
04h	00h or 01h	set an otherwise unused flag to 00h or 01h respectively
04h	02h or 03h	set (another) otherwise unused flag to 00h or 01h, respectively, but also flush the cache
05h	word	set the number of timer ticks between flushes to the value supplied (but current versions of SMARTDrive do not proceed with the installation of the int 1Ch handler, thereby rendering this function ineffective)

Table 13-2: Continued

COMMAND	DATA	DESCRIPTION
06h	-	flush cache and mark current contents as non-discardable
07h	-	mark all cached tracks as discardable
08h	00h or 01h	disable or enable respectively the facility for flushing the cache on int 19h
09h	-	unused (genuinely!)
0Ah	00h or 01h	set (yet another) otherwise unused flag to 00h or 01h, respectively
0Bh	word	reduce cache by the designated multiple of 16K. Inability to resize the cache's memory allocation produces the General Failure error, a do attempts to reduce the cache by more than its current size.
0Ch	word	increase cache by the designated multiple of 16K. Inability to resize the cache's memory allocation produces the General Failure error, but attempts to increase the cache to more than its maximum configured size are simply truncated.
0Dh	dword	divert the int 13h chain below SMARTDRV to the given address (pointer to the int 13h handler currently below SMARTDRV is returned in the IOCTL Read structure)

As well as having programs other than Windows manipulate SMARTDrive for their own benefit, the scheme may be worked in reverse to allow other disk-caching programs to cooperate with Windows by offering an IOCTL interface modelled on SMARTDrive. An example is Compaq's CACHE.EXE, whose primary device name is "CACHCMPQ" with its own IOCTL interface, but which also provides driver routines for a "SMARTAAR" device—mimicking SMARTDrive just well enough that Windows resizes the cache without knowing the difference.

SMARTDrive's IOCTL interface supports more than Windows' straightforward minimization of the cache at startup and restoration on exit. The mechanics of SMARTDrive's disk-caching are not discussed in this book but have been examined fairly thoroughly in the article "Untangling SMARTDrive" (*Dr. Dobb's Journal*, January 1992), from which both the preceding tables have been adapted.

SMARTDRV.EXE (SMARTDrive 4.0) Interface

The SMARTDRV.EXE disk cache supplied with Windows 3.1 does not implement the IOCTL interface of its predecessor and, indeed, does not provide any interface through which programs other than Windows may safely resize the cache. Given that this book has discussed the old SMARTDRV.SYS only because it is *usable* to DOS programmers (who can steal extended memory from it) and as an example of a real-mode program that uses extended memory dynamically (unlike RAMDRIVE, whose allocation is fixed at initialization), SMARTDRV.EXE fits in only because Microsoft would rather it replace SMARTDRV.SYS. Having discussed the old SMARTDRV.SYS, readers will likely expect some discussion of the new SMARTDRV.EXE as well.

While SMARTDRV.EXE does not provide an interface for resizing the cache, it will cut back its cache (thereby releasing memory) when Windows starts up; it attempts to restore the original cache size when Windows exits. Thus, programs that wish to resize the SMARTDRV.EXE cache would (if this idea weren't unsafe for other reasons) need to pretend to be Windows by issuing the int 2Fh function 1605h and 1606h Windows initialization and termination broadcasts. On receiving the 1605h, SMARTDRV.EXE will reduce its cache.

Instead of the old SMARTDRV.SYS IOCTL interface, SMARTDRV.EXE uses int 2Fh function 4A10h. As seen in Table 13-3, some of the functions return a signature hexadecimal number 0BABEh in AX; not surprisingly, the interface is sometimes referred to as the BABE interface.

Table 13-3: Int 2Fh Support for SMARTDRV.EXE Version 4.0, Distributed with Windows 3.1

SMARTDRV Function 00h - Installation Check and General Query

INPUT

AX	4A10h
BX	0000h

OUTPUT

ax	BABEh
dx:bx	number of cache hits
di:si	number of cache misses
cx	number of dirty elements
bp	version number - major version in high byte

SMARTDRV Function 01h - Commit Cache

INPUT:

AX	4A10h
BX	0001h

This function implements the **smartdrv /c** switch.

Table 13-3: Continued

SMARTDRV Function 02h - Restart Cache

INPUT:

AX	4A10h
BX	0002h

This function implements the **smartdrv /r** switch.

SMARTDRV Function 03h - Get/Set Cache Status

INPUT

AX	4A10h	
BX	0003h	
BP	0-based DOS drive number	
DL	00h	to get status
	01h	to enable read cache
	02h	to disable read cache
	03h	to enable write cache
	04h	to disable write cache

SUCCESS

ax	BABEh
dl	unit number of drive with respect to real device driver masked by 3Fh
	bit 6 set if write cache disabled
	bit 7 set if read cache disabled

FAILURE

ax	BABEh
dl	FFh if drive is not physical, that is, the 0-based drive number is ≥ number of DPBs.

When read-caching is disabled, the drive is not listed in the SMARTDRV statistics.

Table 13-3: Continued

SMARTDRV Function 04h - Get Cache Size

INPUT

AX	4A10h
BX	0004h

OUTPUT

ax	number of elements in full cache
bx	number of elements currently in cache
cx	size of element in bytes
dx	number of elements in cache when Windows is running

SMARTDRV Function 05h - Get Double-buffering Status

INPUT

AX	4A10h
BX	0005h

OUTPUT

ax	BABEh	
es:di	address of 16-byte array representing successive fixed disks supported by int 13h	
	00h	double-buffering reported as "-"
	FFh	double-buffering reported as "yes"
	else	double-buffering reported as "no"

The SMARTDRV program's understanding that only 00h represents an uncertain condition is not strictly justifiable. Unless the double-buffering device driver is configured with the **/d+** switch (to force double-buffering in all circumstances), the need for double-buffering is established dynamically. The double-buffering driver counts the number of successful reads from disk to an upper memory address rejected by the VDS server as discontiguous. When the value 04h is reached, double-buffering is deemed unnecessary for the drive. Thus, values 00h to 03h (rather than just 00h) mean that the position is still not clear.

During SMARTDRV's initialization, this call is *issued* for each drive to determine whether or not SMARTDRV should set a default-caching status. This could be used to lend SMARTDRV-defensiveness to disk compressors that SMARTDRV does not know about. Without this facility, the user must supply SMARTDRV with the necessary command line to prevent caching a compressed volume.

Table 13-3: Continued

SMARTDRV Function 06h - Check Drive Cacheability

INPUT

AX	4A10h
BX	0006h
CX	1-based DOS drive number

OUTPUT

ax	0006h to direct SMARTDRV (when loaded later) to disable caching as the default for this d

SMARTDRV Function 07h - Get Original Driver Address

INPUT

AX	4A10h
BX	0007h
BP	1-based DOS drive number

SUCCESS

dl	unit number of drive with respect to its device driver
es:di	address of device driver

SUCCESS

dl	3Fh if the drive is non-physical
es:di	corrupt

SMARTDRV works by creating new device driver headers for all drives known to DOS. In SMARTDRV's scheme, unit numbers and zero-based DOS drive numbers are equal. This function gives access to the device driver ultimately responsible for the given drive *and* the unit number by which that device driver recognizes the drive.

The contents of a cache element are identified by a 32-bit value. The low 24 bits describe the start of the cached region on disk, measured in elements but offset to synchronize element boundaries with the first data sector on disk (and therefore with cluster boundaries). The high 8 bits are the zero-based DOS drive number.

Table 13-3: Continued

SMARTDRV Function 0Ah - Get Element Status Table

INPUT

AX	4A10h
BX	000Ah

OUTPUT

es:bx address of table

OFFSET	SIZE	DESCRIPTION
00h	word	offset within SMARTDRV of byte array (if element size < 2000h) or word array (if element size ≥ 2000h)
02h	word	offset within SMARTDRV of byte array (if element size < 2000h) or word array (if element size ≥ 2000h). Each byte or word is a bit string of dirty sectors within the corresponding element (set if dirty).
04h	word	offset within SMARTDRV of word array. Each word is the low half of a unique identifier for the contents of the corresponding element.
06h	word	offset within SMARTDRV of word array. Each word is the high half of a unique identifier for the contents of the corresponding element.
08h	word	offset within SMARTDRV of word containing number of elements currently in cache

SMARTDRV Function 1234h - Display Serious Error Message

INPUT

AX	4A10h
BX	1234h

Function 1234h invokes SMARTDRV's serious disk-error message harmlessly. The message in this case is for drive D. In a real error situation, 'D' is replaced by the relevant drive, if known, or by the character '!'. The message routine polls the keyboard for the 'r' key.

The undocumented switch **/y** directs SMARTDRV not to commit write-behind data to disk as its normal response to int 21h function 68h

The undocumented switch **/n**: allows direct specification of the number of elements for the full cache when loading SMARTDRV for the first time, but seems equivalent to the **/r** switch if SMARTDRV is already loaded.

DOS Reentrancy

Readers who have heard how DOS is not re-entrant may wonder how XMSFILE's device-driver code (executing as part of an int 21h function) can even contemplate issuing IOCTL commands to another device driver; clearly, this cannot be done by calling DOS functions from inside DOS!

One solution is for XMSFILE to call SMARTDrive's strategy and interrupt routines directly, passing appropriate request headers for the IOCTL functions. If SMARTDrive supported functions to open and close its logical device, these too would have to be called, as would happen if accessing the device through int 21h functions. The problem with this solution is that by undercutting the DOS kernel, it skips the file system's support for devices and, in particular, it by-passes controls on file sharing. This may be seen easily enough by executing the PRINT utility (which manages its output device by direct access to the corresponding driver), having first opened the PRN device in a restrictive sharing mode. Readers with a copy of SYMDEB will be able to assemble the necessary instructions in free memory and then shell out to DOS at just the right moment with the handy **!** command. This debugger's more sophisticated and apparently more useful replacement (CodeView) insists on having a program to debug and is less amenable to casual hypothesis testing. Consider the short program

```
.MODEL              SMALL, FARSTACK ; for brevity in .STARTUP code
.STACK              128
.DATA
prn                 db      "prn", 00h
.CODE
.STARTUP                    ; sets ds = DGROUP
                    mov     dx,OFFSET prn
                    mov     ax,3D12h  ; read/write, deny all sharing
                    int     21h
.EXIT                                 ; calls DOS function 4Ch
END
```

Call this something like TEST.ASM, then assemble and link. To ensure that enough memory will be available for the debugger to run DOS commands, use the EXEHDR utility, as in

```
exehdr /max:0 test.exe
```

so that the program will receive only the smallest possible memory allocation at load-time. Incidentally, it is not intended to set a maximum memory requirement of 0 paragraphs as the above command suggests, but rather to rely deliberately on EXEHDR's documented correction facility. Load the resulting executable into CodeView, step through to the instruction after the int 21h that opens the PRN device, and shell out to the DOS prompt. The command

```
type test.asm >prn
```

triggers the expected sharing violation (assuming here that SHARE has been loaded), but

```
print /d:prn test.asm
```

succeeds. File-sharing operates at DOS-function level and is frustrated by programs like PRINT that work with device drivers directly. In the absence of other solutions, possible confusion over file-sharing might not seem a substantial risk compared to the benefit of gaining extra memory through the IOCTL calls to SMARTDrive.

The DOS Kernel's Sidedoor Interface

A better solution exists however, because this problem of managing a file at DOS-function level from code that is already processing a DOS function is exactly the problem faced by the NLSFUNC utility when extracting information from COUNTRY.SYS for int 21h functions 38h (Get/Set Country Information), 65h (Get Extended Country Information, etc.), and 66h (Get/Set Global Code Page).

The DOS kernel supports a number of possible extensions—optional system components that act beneath the int 21h interface to enhance or even replace the normal processing of DOS functions. NLSFUNC is a comparatively recent example, dating from DOS 3.30. Before then, country-specific information (date and time formats, currency symbols, etc.) was hard-coded into the MSDOS system file, but improvement of DOS's national language support brought a significant growth in the amount of data involved (upper-case maps, collate sequences, etc.) for each locale, best met by shifting the relevant tables into a separate file (COUNTRY.SYS for instance) and providing an optional component (NLSFUNC.EXE) to link the data with the kernel. Users with no need for international support could simply ignore the files—along with a substantial tract of the DOS manual—and know that the system was not wasting valuable memory on anything unAmerican.

At the programming level the kernel code for int 21h functions 38h, 65h, and 66h can deal with requests involving the country and code page configured at system startup, but anything more complicated requires support from NLSFUNC. A country-specific problem that cannot be handled by default is passed from the kernel to the optional system component NLSFUNC in much the same way that hardware-specific problems are sent to device drivers. Code in NLSFUNC must be able to open COUNTRY.SYS, move the file pointer, read data, and close the file even though this code is executing as part of an int 21h function. To support this sort of activity by system extensions that the kernel calls directly, DOS provides a mechanism through which it may be called back—and this is how XMSFILE's device driver routines will reenter DOS to open and manipulate the SMARTAAR device.

Lest there exist readers to whom DOS's provision for internationalization is not an issue of burning importance, rest assured that what is really being addressed here is the issue of making DOS calls while an int 21h call is already in progress. The notion of extending the kernel's functionality through optional components is not limited to NLSFUNC. The SHARE utility supplies extra kernel code that monitors file operations more closely and offers protection against simultaneous access. The redirector interface allows replacement of the default file system, initially to support networks but now applied more generally to drives set up by other operating systems or whose physical characteristics make the 12- and 16-bit FAT systems unfeasible. The FASTOPEN program caches directory entries to help the kernel find files faster. All these act as extra modules to the DOS kernel, ensuring efficient and robust integration of their additional functionality with the kernel's standard behavior.

When the extensions are called from the DOS kernel to help process an int 21h function, conditions are very much the same as those for code in the kernel itself. An internal DOS stack is in use and kernel data is addressable with respect to the SS register. From the perspective of design, offset addresses for data variables in the kernel can be built into an extension at link time since the amount of data is relatively small. This is why SHARE's executable file ends with an image of the kernel's data area. Offset addresses for useful routines in the kernel cannot be known so easily, there being far too much kernel code to link. Instead, DOS makes a selection of kernel routines available to its system extensions via an int 2Fh function 12h interface. Most of the functions provided this way are for redirectors and SHARE—to promote uniform treatment of filename parsing, critical error handling, and so on, or to hide the ever-changing **buffers** system behind a stable interface. In the absence of system extensions, these routines would be called as a normal part of int 21h processing, but the ones of interest here actually duplicate whole int 21h functions. As already noted, they were introduced when national language support was separated from the kernel.

When the NLSFUNC extension is loaded, the DOS kernel may pass responsibility for int 21h functions 38h, 65h, and 66h to code in NLSFUNC via int 2Fh function 14h. In turn, NLSFUNC reenters the kernel by using int 2Fh function 12h when it needs the kernel's assistance to read the COUNTRY.SYS file or to send IOCTL commands to device drivers (when updating the global code page). To each int 21h function that would normally be used for such file operations there corresponds a function in the sidedoor interface, as shown in Table 13-4. Observe that sidedoor functions take the same input as their ordinary counterparts, except for the different use of register AX. The sidedoor interface uses AL to distinguish functions and must therefore provide another register for values that would ordinarily have been placed in AL.

When reentering the DOS kernel, the sidedoor interface bypasses the usual dispatch code of a DOS function. There is no stack-switching (a DOS stack must be in use already), no resetting of global kernel variables or critical section signals, and so on. Most strikingly, there is no explicit preservation of registers, which is a small complication in what would otherwise be a mechanical conversion from using int 21h functions.

Table 13-4: Duplication of Int 21h Functions for NLSFUNC's Access to COUNTRY.SYS

DOS FUNCTION - AH FOR INT 21H	SIDEDOOR FUNCTION - AX FOR INT 2FH	SUBSTITUTE REGISTER FOR VALUE NORMALLY PASSED IN AL
3Dh (open file)	1226h	CL
3Eh (close file)	1227h	
3Fh (read from file)	1229h	
42h (move file pointer)	1228h	low byte of BP
44h (IOCTL etc)	122Bh	low byte of BP

In the following file, routines written to control SMARTDrive using int 21h functions are easily adapted for operation through the DOS kernel's sidedoor interface. Note that the module sometimes refers to the sidedoor interface using the old name "callback," which the author had worked with temporarily until realizing that "sidedoor" was more a more accurate description.

```
/* smartdrv.c */

/*  This file (which may be compiled as a resident module under the Keep C
    scheme for resident programming) contains routines for manipulating the
    SMARTDrive disk cache, both for small steps (open, read, write, close)
    and for the particular task of resizing the cache in extended memory.

    To adapt the routines for use when control has already passed through
    the DOS kernel, i.e., to demonstrate the undocumented sidedoor facility
    provided by int 2Fh function 12h, define the symbol _CALLBACK_ at
    compile time and use the segmentation model for resident programming.
    The term _CALLBACK_ is merely an artifact; "sidedoor" is accurate. */

#include    <driver.h>
#include    <keepc.h>
#include    <sft.h>
#include    <smartdrv.h>
#include    <standard.h>

/*  The logical device name by which SMARTDrive is known to DOS */

static CHAR smartaar [] = "SMARTAAR";

/*  Transfer buffers for SMARTDrive control data. */

static struct SD_READ sd_read = {0};
static struct SD_WRITE sd_write = {0};

/*  For the convenience of external modules, separate functions are provided
    to make the SMARTDrive cache grow or shrink (by some number of K). */

BOOL DecreaseCache (DWORD);
BOOL IncreaseCache (DWORD);

/*  Both functions behave similarly in terms of their access to
    SMARTDrive, so they are passed to a single function that carries a
    variable (the SMARTDrive command number) to distinguish expansion from
    contraction. */

static BOOL ChangeCache (BYTE, DWORD);

/*  Separate C functions for each call to the SMARTAAR device. These are
    written with in-line assembler to demonstrate the correspondence between
    the int 21h functions and their sidedoor equivalents. Registers may be
    corrupted during the DOS call, but as written here, these C functions
    preserve only ds and bp explicitly—the liability of si and di to
    change is fairly easily accommodated by the caller, as shown by the
    design of ChangeCache (). */

int OpenSMARTDrive (void);
BOOL SupportsIOCTL (int);
BOOL ReadSMARTDrive (int, struct SD_READ *);
BOOL WriteSMARTDrive (int, struct SD_WRITE *);
BOOL CloseSMARTDrive (int);

#ifdef  _CALLBACK_

/*  Special preparation required for the sidedoor interface.*/

static BOOL IsCallbackOK (void);

/*  For strict safety, some material in the DOS kernel's data area must be
    preserved across the call to SMARTDrive. Otherwise, when the kernel
    opens the SMARTAAR device, it will lose the address of the SFT it has
    opened for the XMSFILEX device. */

static void SaveKernelData (void);
static void RestoreKernelData (void);
struct SFT _far * _far *lpSFTPointer = NULL;

extern struct REQUEST_HEADER _far *lpRequestHeader;      // in driver.asm
```

```
#endif

/* ====================================================================== */

BOOL IncreaseCache (DWORD increment)
{
    return (ChangeCache (SD_INCREASE_CACHE, increment));
}

BOOL DecreaseCache (DWORD decrement)
{
    return (ChangeCache (SD_DECREASE_CACHE, decrement));
}

/* ---------------------------------------------------------------------- */

static BOOL ChangeCache (BYTE command, DWORD change)
{
    register WORD change_16K, latitude;        // register keyword ensures
    int handle;                                // preservation of si and di
    BOOL cache_changed = FALSE;

    /*  The SMARTDrive cache size is measured and manipulated in multiples
        of 16K, using a 16-bit variable, so check that the desired change
        (supplied in K) is within range. */

    if (HIGHWORD (change) >> 4) return (FALSE);

    /*  The sidedoor facility assumes an internal DOS stack. check that
        the current stack belongs to the DOS kernel. */

    #ifdef  _CALLBACK_
    if (NOT IsCallbackOK ()) return (FALSE);
    SaveKernelData ();
    #endif

    /*  Open the SMARTAAR device. */

    handle = OpenSMARTDrive ();
    if (handle == -1) return (FALSE);

    /*  Check that the resulting handle does in fact refer to a character
        device that supports IOCTL, then read SMARTDrive's configuration. */

    if (SupportsIOCTL (handle) AND ReadSMARTDrive (handle, &sd_read)) {
         /*  If the cache is in extended memory, calculate the increment
             or decrement in multiples of 16K and verify that SMARTDrive
             is able to accommodate the change. */

         if (sd_read.MemoryType == 0x01) {

             change_16K = (HIGHWORD (change) << 12) + (LOWWORD (change) >> 4);
             if (change_16K != 0) {

                   latitude = command == SD_DECREASE_CACHE ?
                             sd_read.CurrentSize - sd_read.MinimumSize :
                             sd_read.MaximumSize - sd_read.CurrentSize;

                   if (change_16K <= latitude) {

                       /*    Send the desired cache change to SMARTDrive via the
                             IOCTL Write interface. */

                       sd_write.command = command;
                       sd_write.data.size = change_16K;

                       if (WriteSMARTDrive (handle, &sd_write))
                             cache_changed = TRUE;
                   }
             }
         }
    }
    CloseSMARTDrive (handle);

    #ifdef  _CALLBACK_
    RestoreKernelData ();
    #endif

    return (cache_changed);
}

/*  ------------------------------------------------------------------------  */

#ifdef   _CALLBACK_

static BOOL IsCallbackOK (void)
{
    _asm {
```

```
    /*  Return TRUE if and only if the current stack belongs to the DOS
        kernel. The segment address of the kernel's data and stack is
        returned in register ds by int 2Fh function 1203h (one of the
        few sidedoor services that do not assume an internal DOS stack). */

            push    ds
            push    bp
            mov     ax,1203h
            int     2Fh
            mov     ax,ds
            mov     dx,ss
            pop     bp
            pop     ds
            cmp     ax,dx
            jnz     bad

            mov     ax,0001h        ; TRUE
            jmp     done
        bad:
            xor     ax,ax   ; FALSE
        done:
    }
}

#endif

int OpenSMARTDrive (void)
{
    _asm {

    /*  Open the SMARTDrive character device (SMARTAAR) for read/write
        access, returning the handle if successful, or else FFFFh. Sidedoor
        function 26h invokes the same code used by DOS function 3Dh to open
        a file. */

            mov     dx,OFFSET smartaar

    #ifndef _CALLBACK_
            mov     ax,3D02h
            int     21h
    #else
            mov     cl,02h ; access flags in cl instead of al
            push    ds
            push    bp
            mov     ax,1226h
            int     2Fh
            pop     bp
            pop     ds
    #endif
            jnc     done
            mov     ax,0FFFFh
        done:
    }
}

BOOL SupportsIOCTL (int handle)
{
    _asm {

    /*  Return TRUE if and only if the handle refers to a character device
        that responds to IOCTL. The whole int 21h function 44h family is
        supported by sidedoor function 2Bh - the subfunction number usually
        in AL is passed in the low byte of BP instead. */

            mov     bx,handle

    #ifndef _CALLBACK_
            mov     ax,4400h            ; subfunction 00h - Get Device Status
            int     21h
    #else
            push    ds
            push    bp
            mov     bp,4400h
            mov     ax,122Bh
            int     2Fh
            pop     bp
            pop     ds
    #endif
            jc      bad
            and     dx,4080h            ; 0080h is mask for character device,
            cmp     dx,4080h            ; 4000h for IOCTL SUpport
            jnz     bad
            mov     ax,0001h            ; TRUE
            jmp     done
            bad:
            xor     ax,ax   ; FALSE
        done:
    }
}
```

```
BOOL ReadSMARTDrive (int handle, struct SD_READ *sd)
{
    _asm {

    /*   Return TRUE if and only if an IOCTL Read from SMARTDrive (of the
         whole configuration structure) succeeds. */

            mov     dx,sd
            mov     cx,size sd_read
            mov     bx,handle

    #ifndef _CALLBACK_
            mov     ax,4402h        ; subfunction 02h - IOCTL Read from device
            int     21h
    #else
            push    ds
            push    bp
            mov     bp,4402h
            mov     ax,122Bh
            int     2Fh
            pop     bp
            pop     ds
    #endif
            jc      bad
            cmp     ax,cx
            jnz     bad
            mov     ax,0001h        ; TRUE
            jmp     done
            bad:
            xor     ax,ax   ; FALSE
        done:
    }
}

BOOL WriteSMARTDrive (int handle, struct SD_WRITE *sd)
{
    _asm {
            mov     dx,sd
            mov     cx,size sd_write
            mov     bx,handle

    #ifndef _CALLBACK_
            mov     ax,4403h        ; subfunction 03h - IOCTL Write to device
            int     21h
    #else
```

```
            push    ds
            push    bp
            mov     bp,4403h
            mov     ax,122Bh
            int     2Fh
            pop     bp
            pop     ds
    #endif
            jc      bad
            cmp     ax,cx
            jnz     bad
            mov     ax,0001h          ; TRUE
            jmp     done
            bad:
            xor     ax,ax   ; FALSE
        done:
    }
}

BOOL CloseSMARTDrive (int handle)
{
    _asm {

    /*  Close the file handle, using DOS function 3Eh as usual or sidedoor
        function 27h. */

            mov     bx,handle

    #ifndef _CALLBACK_
            mov     ah,3Eh
            int     21h
    #else
            push    ds
            push    bp
            mov     ax,1227h
            int     2Fh
            pop     bp
            pop     ds
    #endif
            jc       bad
            mov     ax,0001h          ; TRUE
            jmp     done
            bad:
            xor     ax,ax   ; FALSE
        done:
    }
}

/* ------------------------------------------------------------------- */

#ifdef      _CALLBACK_

/* Temporary storage for the kernel's pointer to the SFT opened for the
   XMSFILEX device  */

static struct SFT _far *lpXMSFILEX_SFT = 0;

/* The REQUEST_HEADER structure contains unnamed substructures and cannot
   easily be declared as initialized data, hence the following ugly
   construction for a temporary area in which to save the device driver
   header passed to the XMSFILEX device while asking DOS to call
   SMARTDrive. */

static BYTE dummy [sizeof (struct REQUEST_HEADER)] = {0};
#define XMSFILEX_ReqHdr *((struct REQUEST_HEADER *) dummy)

void SaveKernelData (void)
{
    lpXMSFILEX_SFT = *lpSFTPointer;
    XMSFILEX_ReqHdr = *lpRequestHeader;
}

void RestoreKernelData (void)
{
    *lpSFTPointer = lpXMSFILEX_SFT;
    *lpRequestHeader = XMSFILEX_ReqHdr;
}

#endif
```

Once SMARTDrive has been persuaded to release some of its memory, XMSFILE may be able to add the new memory to its own EMB and resume the original device-driver request. Hopefully, even if readers haven't yet perused the SMARTDRV.C module, they have wondered whether all the kernel activity involved in executing nested DOS functions through the sidedoor interface changes any internal DOS variables that the kernel needs to finish the original DOS call. Possible corruption of this sort is not a concern when the locale-oriented DOS functions call NLSFUNC to consult the COUNTRY.SYS file, because those functions have nothing else to do with the file system.

The obvious conflict when one device driver uses the sidedoor interface to access a second device driver is that DOS passes the same request header to both device drivers. The real difficulty lies with data saved for the kernel's own bookkeeping purposes and not passed to device drivers. In the case at hand, the kernel is first asked via a DOS function to work with XMSFILEX, say to write some data to the device (which may entail asking SMARTDrive for memory) or to close the device (at which time, any memory taken from the disk cache must be returned). In response to this, the kernel saves a pointer to the System File Table (SFT) for the open XMSFILEX device. The SMARTDrive operations cause the kernel to switch its attention to an SFT it creates for SMARTAAR and to lose the address of the SFT for XMSFILEX.

After XMSFILEX has dealt with the device driver request, DOS may need to look again at the SFT. In many situations, DOS will have kept the SFT address on the stack while calling the device driver and in the case of the console I/O functions which imply a particular handle with respect to the current process, DOS may actually find the SFT address afresh by dereferencing through the PSP. In other cases, it takes the SFT address from a variable dedicated as a pointer to the "working" SFT, but this will now point to memory that contained the SFT for the SMARTAAR device.

Because of this predicament, the SMARTDRV.C file provides not only an alternative coding of functions that manipulate the disk cache, but also a couple of functions that save and restore this internal variable—the DOS kernel's pointer to the working SFT.

`SaveKernelData` and `RestoreKernelData` in SMARTDRV.C use an `lpSFTPointer` variable. This is initialized by the `InitSMARTDrive` routine in SDINIT.C. This routine accesses DOS kernel data using the same technique that several utilities shipped with DOS use: It calls int 21h function 52h (Get SysVars) but, rather than using the SysVars pointer returned in ES:BX, it ignores BX and takes ES as the DOS data segment. Depending on the value at DOS:[4] it then uses one of two offsets.

Direct Access to Kernel Data

Programmers should not view direct access to kernel data as anything other than a last resort. A program's byte-by-byte internal organization is highly susceptible to change, not only from one version to another, but also within a release.

Against this, for designing kernel extensions and particularly for redirectors, there is little choice but to address kernel data directly. The redirector (int 2Fh with AH=11h) and side-door interfaces (int 2Fh with AH=12h) let control flow in and out of the DOS kernel but help very little with access to data. The ill-fated IFSFUNC program in DOS 4 attempted to deal with this by converting these interfaces into a device-driver type of exchange through data packets, but this reorganization appears to have been unwanted (probably for performance reasons); precisely because the redirector interface forces clients to access data in the DOS kernel directly, the addresses have rather more stability across DOS versions than might ordinarily be expected.

A good example is offered by the MSCDEX program—a redirector for the High Sierra and ISO-9660 file systems used by CD-ROMs. Unlike other kernel extensions such as SHARE and REDIR, this program is not tied to a single DOS version. That MSCDEX 2.21 can reasonably expect to cope with all DOS versions from 3.10 up to but not including 6.00 is because DOS kernel data exists in only two layouts. The idea of categorizing the arrangement of kernel variables (or at least the interesting ones) appears to have been built-in from the start. Microsoft programs that use kernel data directly check not only the DOS version but also the word at offset 04h in the kernel's data segment. This word is 0000h if variables are laid out in the style of DOS 3.xx but is 0001h to denote changes introduced for DOS 4. DOS 5 adheres closely to the arrangement of its immediate predecessor for all material up to the end of what is usually known as the Swappable Data Area; for DOS 5 and 6 also, the data format at offset 04h has the value 0001h.

The means by which Microsoft's programs locate the DOS kernel for the purpose of direct access to its data is through int 21h function 52h, which returns with es:bx addressing the SYSVARS structure. A documented interface that returns an address in the kernel's data area is int 21h function 34h—for the InDOS flag.

The latter method is recommended—in a different context—by the MASM 6.0 manual in its section on writing resident programs. The different objective is to locate the critical error flag—easy enough in DOS versions 3.1 and higher, because it is always the byte immediately before the InDOS flag. Apparently this relationship does not hold for earlier DOS versions or the OS/2 compatibility box. The MASM manual recommends searching the segment containing the InDOS flag for a specific sequence of instructions that issue int 28h, since this is preceded by a check that the critical error flag is clear. Interestingly, the separation of kernel code and data for DOS 5 means that the data segment contains quite a few instruction sequences that software is known to search for in various versions of DOS kernel.

As an aside to this aside, note that the critical error flag's location ahead of the InDOS flag is relied on by both PRINT and DOSSWAP, and is documented in the official reference; clearly this is Microsoft's preferred way of locating the critical error flag. Texts on DOS occasionally describe int 21h function 5D06h as returning the address of the critical error flag. This interpretation must be regarded as suspect. The function returns the size and location of two overlapping regions of kernel data, either or both of which must be preserved by code that interrupts the main flow of control and intends to issue DOS functions. There is no evidence that DOS is written with these swappable data areas as structures, much less that they must begin with the critical error flag. While history shows that once someone has put even a groundless observation about DOS into print or incorporated it into sufficiently popular software, Microsoft—to their credit—support it if possible, this is no reason to perpetuate such practices.

The use of one device driver from inside another produces a neat example of how to use kernel data without letting the technique run out of proportion. Just one variable's address must be determined during XMSFILE's initialization and recorded in a resident module for use whenever memory is needed from SMARTDrive:

```
/* sdinit.c */

/*  This file performs some initialization specific to using the
    SMARTDRV.SYS disk cache as a source of additional memory for the XMSFILE
    clipboard. */

#include    <stdlib.h>

#include    <keepc.h>
#include    <sft.h>
#include    <standard.h>

/*  The address of a variable in the DOS kernel data area—needed if
    accessing the SMARTDrive cache's device while already executing an
    int 21h function */

extern struct SFT _far * _far * _RESIDENT lpSFTPointer; // in smartdrv.c

extern void PutWarning (CHAR *, ...);                         // in main.c

BOOL InitSMARTDrive (WORD clustersizeinkb)
{
    register _segment KernelSegment;
    register WORD KernelDataFormat;

    /*    Check that additional memory will be sought in multiples of the same
          unit (16K) that SMARTDrive uses for resizing its memory allocation.
          It is assumed here that the clustersizeinkb argument is already
          known to be a power of 2. */

    if (clustersizeinkb < 16) {
          PutWarning ("clustersize too small to support use of SMARTDrive");
          return (FALSE);
    }

    /*    For safe use of SMARTDrive while already processing DOS functions,
          obtain the address of the DOS kernel variable that must be
          preserved. */

     _asm {
               mov       ah,52h
               int       21h
               mov       KernelSegment,es  /* ignore bx */
     }
     KernelDataFormat = *((WORD _far *) MK_FP (KernelSegment, 0x0004));

     if (KernelDataFormat != (WORD) (_osmajor < 4 ? 0 : 1)) {
          PutWarning ("unrecognized DOS version - cannot use SMARTDrive");
          return (FALSE);
     }

     lpSFTPointer = (struct SFT _far * _far *)
               MK_FP (KernelSegment, KernelDataFormat == 0 ? 0x0536 : 0x059E);

     return (TRUE);
}
```

The Completed XMSFILE

Yet again, a makefile is almost essential (even in this fairly small project) to ensure that resident modules are compiled with the required switches and that the device-driver front-end module be linked first. While this makefile assumes that everything needed is in the current directory, the version on disk accounts for the use of modules such as SEQFILE.C from other projects.

```
!INCLUDE <standard.mak>
!INCLUDE <keepc.mak>

TARGETS : xmsfile.exe

driver.obj : driver.asm
 $(ASSEMBLE) /D "_ATTRIBUTE_=IS_CHAR_DEVICE OR IOCTL_OK OR OCRM_OK" \
 /D _NAME_=XMSFILEX driver.asm

main.obj : main.c

sdinit.obj : sdinit.c

seqfile.obj : seqfile.c
 $(COMPILE) $(KEEP) seqfile.c
```

```
smartdrv.obj : smartdrv.c
 $(COMPILE) $(KEEP) /D_CALLBACK_ smartdrv.c

transfer.obj : transfer.c
 $(COMPILE) $(KEEP) transfer.c

#  Link with DRIVER.OBJ, CRTDRVR.LIB and CRTKEEPC.LIB

xmsfile.exe : driver.obj main.obj sdinit.obj seqfile.obj smartdrv.obj \
              transfer.obj
 $(LINK) driver+crtdrvr.lib+crtkeepc.lib+getxms+ \
 main+sdinit+seqfile+smartdrv+transfer,xmsfile.exe;
```

Add the appropriate **device** or **devicehigh** directive in CONFIG.SYS, then reboot. XMSFILE's footprint in conventional or upper memory is 3K, in return for which it offers a clipboard that tailors its extended memory usage to fit immediate need. The clipboard is best operated in binary mode. Support for binary mode access to character devices from the DOS command line is fairly primitive, but text files may be copied into the clipboard and retrieved easily enough by using the **type** command, as discussed in Chapter 7 for the int 15h version on which XMSFILE is meant to improve.

Unlike its predecessor, XMSFILE has no fixed capacity. When it is inactive, it releases its unneeded memory, and when additional storage space is required, the XMS server may even be able to move the clipboard contents into a larger block of unused memory if the current block is too small to accommodate expansion. How, though, is one to witness the flexibility of XMS memory management? The MEM utility provided with DOS 5 displays only the total amount of extended memory and the amount of XMS memory still available (as noted at the end of Chapter 11, the version of MEM included with DOS 6 does use HIMEM.SYS int 2Fh function 4309h to list XMS handles, but this is hardly a general solution). The next chapter presents an XMSWATCH utility.

CHAPTER 14

Extending the XMS

When it comes to obtaining a "map" of extended memory usage, the XMS interface is especially unhelpful, for although free EMBs can be locked and mapped, information about memory blocks already allocated can be obtained only if one possesses the corresponding XMS handles. This is far from trivial. The XMS interface provides no function for enumerating handles that have been allocated already.

In the specific case of HIMEM.SYS, Chapter 11 noted that version 3.09 introduces an extension to the XMS int 2Fh interface which enables programs to inspect HIMEM's internal list of XMS handles. But even if the facility were documented, it would be of relatively little use given that the XMS server need not be HIMEM and that, in any case, many machines running HIMEM will not have the most recent version. Moreover, Chapter 13 has already noted that task switchers manufacture XMS handles of their own so that they can control the XMS environment seen by tasks. Int 2Fh function 4309h looks much more like a diagnostic aid specific to HIMEM than a general solution to the problem of mapping extended memory usage.

The only general solution, short of the brute force method of trying all possible 16-bit values to see which ones are recognized as XMS handles, is to interfere with the passage of information between the XMS server and its clients, that is, to snoop on handles at the time of their allocation. Recall that clients call XMS functions through a far pointer rather than with an INT instruction. Thus, snooping on XMS allocations requires an understanding of how to hook XMS.

Hookability

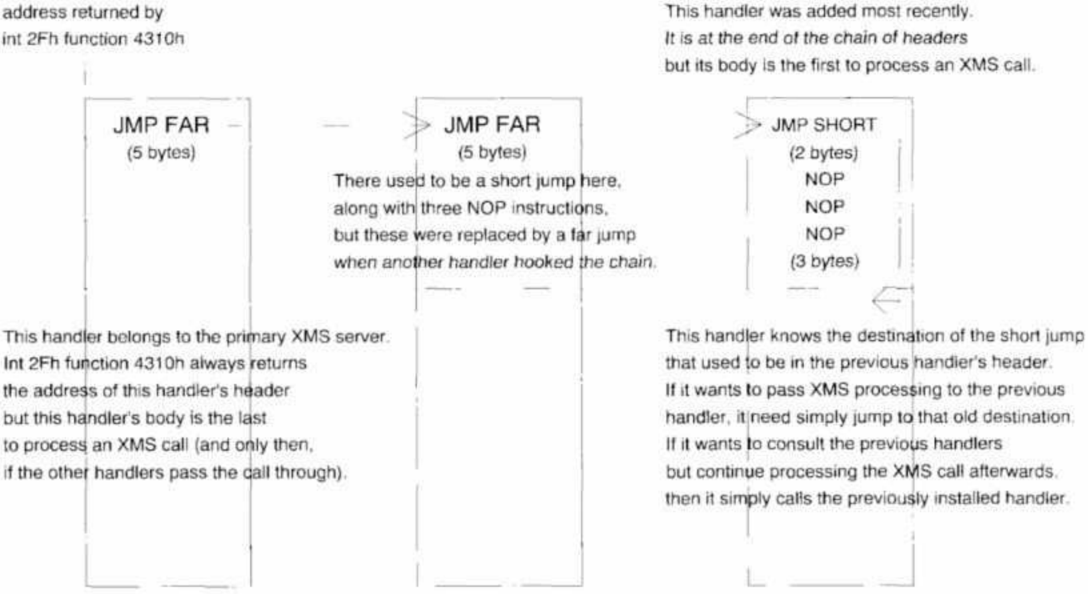
The desire to map extended memory is really just a practical motivation for studying a larger theme. Some inkling of what is still to come was given with the previous chapter's discussion of how an XMS client loaded before a task switcher may have to constrain its use of the XMS to account for the way that the subsequently-loaded task switcher may change the XMS environment. In much the same way that programs hook interrupt vectors in order to modify the behavior of an existing interface, here the intention is that programs, mostly memory managers and task switchers, may want to alter the handling of XMS services.

Hooking an interrupt is very easy because the processor defines the interrupt as a sort of indirect call. Simply by calling int 21h function 25h to change the appropriate entry in the interrupt vector table, software may gain first option on all use of that interrupt in real mode by all other programs, whether they were loaded before or after the software that hooks the interrupt. It is usual for the hooker to save the address of the interrupt handler it has displaced. That way, calls that it does not want to process may be passed onward so that they will be processed as before. Alternatively, the hooker may occasionally want to call the handler it has displaced. It is common to picture a chain of interrupt handlers, starting from the one that has hooked the interrupt most recently and reaching to the one (usually a default installed by the BIOS or by DOS) that was installed first.

But since the XMS interface uses a far function pointer rather than an interrupt, something else is called for. Under the heading "Implementation Notes for DOS XMS Drivers," the Extended Memory Specification provides for a chain of XMS handlers. Each handler is to consist of a five-byte header followed by a body. The header starts with a short jump instruction whose destination lies somewhere in the body. NOP instructions pad the header to the five bytes required for a far jump.

If software knows the address of an XMS handler whose header is still in the form of a JMP SHORT followed by three NOPs, then it may hook control from that XMS handler by replacing the header with a JMP FAR instruction whose destination is the new handler. This way, the new handler gets to process all XMS requests ahead of the handler it has hooked. Figure 15-1 shows how this technique may produce a chain of XMS handlers in which the one added most recently gets first option on all programs XMS calls; just as with hooking an interrupt, this includes XMS calls from programs loaded *after* the most recent XMS handler.

Figure 15-1: Arrangement of XMS Handler Chain



This handler was added most recently.
It is at the end of the chain of headers
but its body is the first to process an XMS call.

There used to be a short jump here,
along with three NOP instructions,
but these were replaced by a far jump
when another handler hooked the chain.

This handler belongs to the primary XMS server.
Int 2Fh function 4310h always returns
the address of this handler's header
but this handler's body is the last
to process an XMS call (and only then,
if the other handlers pass the call through).

This handler knows the destination of the short jump
that used to be in the previous handler's header.
If it wants to pass XMS processing to the previous
handler, it need simply jump to that old destination.
If it wants to consult the previous handlers
but continue processing the XMS call afterwards,
then it simply calls the previously installed handler.

The technique is easily implemented in source code, as shown in the following XMSCHAIN.C, which provides one function to follow successive far jumps, and another function to thread a new handler to the end of the chain of far jumps so that it gains the first look at all XMS requests.

```
/* xmschain.c */

#include  <dos.h>

#include  <standard.h>

#pragma intrinsic (_disable, _enable)

/* An XMS handler must begin with a short jump and three NOP instructions—
  enough padding to allow replacement by an intersegment jump to an
  additional handler. */

struct XMSENTRY {
  BYTE jump;
  union {
          struct {
            SBYTE displacement;
            BYTE nops [3];
          };
          struct XMSENTRY _far *lpNext;
  };
};

#define   NOP       0x90
#define   JMP_FAR  0xEA
#define   JMP_SHORT  0xEB

/* A pointer to the XMS calling point must be initialized before calling
  either function in this module */

extern void (_far *lpXMS) (void);

/* ==================================================================== */

/* Successive handlers are reached by direct intersegment JMP instructions
  patched over the original short JMP and NOPs. The first function in
  this file simply follows the far jump at the start of one handler to
  return the address of the next. */
```

```
struct XMSENTRY _far *FollowXMSChain (struct XMSENTRY _far *callpoint)
{
  if (callpoint -> jump == JMP_FAR) return (callpoint -> lpNext);
  else return (NULL);
}

/* The following function threads a new handler onto the chain, returning
  the address to which XMS requests would have been sent (by a short jump
  before the addition). A null pointer is returned if the chain of far
  jumps is corrupt. */

#pragma optimize ("e", on)

void (_far *ThreadXMSChain (void (_far *newhandler) (void))) (void)
{
  struct XMSENTRY _far *xmsptr;
  void (_far *next) (void);

  /*     Find the end of the chain. */

  xmsptr = (struct XMSENTRY _far *) lpXMS;
  while (xmsptr -> jump == JMP_FAR) {
         xmsptr = xmsptr -> lpNext;
    }

    /*   The last handler must begin with a short jump and three NOP
         instructions. If so, return the destination of the short jump,
         having set in place a far jump to the new handler. */

    if (xmsptr -> jump == JMP_SHORT
         AND *((WORD _far *) xmsptr -> nops) == MK_WORD (NOP, NOP)
         AND xmsptr -> nops [2] == NOP) {

         #define target(x) (&(x) + sizeof (x) + (SWORD)(x))
         next = (void (_far *) (void)) target (xmsptr -> displacement);

         /* Disable interrupts temporarily to guard against the slight
             possibility in complex systems that a hardware interrupt handler
             may issue an XMS call (to test the A20 status, for instance)
             while the far jump is being patched in. */

         _disable ();
         xmsptr -> jump = JMP_FAR;
         xmsptr -> lpNext = (struct XMSENTRY _far *) newhandler;
         _enable ();
```

```
            /* Return the destination of the short jump that has just been
               replaced. The newly-added XMS handler may use this address to
               send XMS requests on their original route or to obtain XMS
               services from the pre-existing server. */

            return (next);
        }
        return (NULL);
    }
```

While the entry point patching shown in XMSCHAIN.C is the officially sanctioned method, it is no good inside a task switcher. The problem lies with the task switcher, which *must* instance the end of the XMS chain and possibly do some further localization on the fly. (The same problem was also encountered in Chapter 2 with the DISKHOOK program, which cannot be run under a task switcher because the int 13h chain is not localized.) This makes a mockery of the XMS documentation. Furthermore, Microsoft's Windows enhanced mode breaks several of the rules laid down for everyone else and therefore complicates any attempt at finding a workable solution. In Windows 3.0, V86MMGR doesn't instance the header it installs in the V86 machine, and the standard-mode task switchers don't instance their XMS headers either, so in practice it's just not safe to run any XMS hooker under a task switcher. Nice one, Microsoft!

XMS hookers must decide whether they want to call just the previously installed handlers or to send out a request to the whole chain (to take in more recently installed handlers as well). Chapter 12 mentioned the HIMEM /Z switch, which changes how HIMEM's once-internal A20 calls are passed down the chain. The particular problem for HIMEM is that it needs to enable/disable the A20 from time to time (to access its own code in the HMA if dos=high, but at the very least as part of XMS function 0Bh, for transferring data to and from extended memory). Until version 3.09, HIMEM just called its own code for the Local Enable/Disable A20 functions, but this presents problems since an XMS hooker installed after HIMEM (e.g., Windows) may depend on tracking the primary XMS server's logical A20 or may have disabled the port accesses that HIMEM would use to enable/disable the A20. The /Z (perhaps Z for Zenith, whose A20 handler might have been especially prone to these effects?) changes the HIMEM behavior so that what used to be internal A20 operations can now be seen by XMS handlers that have hooked the chain.

In any case, an immediate application of the `FollowXMSChain` function in XMSCHAIN.C is to enumerate the installed XMS handlers; XMSWATCH uses this function to print out the XMS chain. The `ThreadXMSChain` function is used later in this chapter when the XMSWATCH program adds its XMS-handle monitor to the chain.

Tracing the XMS Chain

HANDLERS.C displays the XMS handler chain, using the `FollowXMSChain` function.

```
/* handlers.c */

#include     <stdio.h>
#include     <string.h>

#include     <standard.h>

/*  The start of the chain of XMS handlers and a function to trace the chain
    one handler at a time  */
extern void (_far *lpXMS) (void);
extern void (_far *FollowXMSChain (void (_far *) (void))) (void);

/*  A function to identify the owner of each handler in the chain  */
extern CHAR _far * _far GetMemoryOwner (void _far *);

/*  Message header  */
static CHAR header [] = "\n\
\n\
Report on XMS Handler Chain:  \n\
\n\
Entry Point        Owner\n\
------------------------\
";

#pragma optimize ("e", on)

void ReportHandlers (void)
{
    void (_far *callpoint) (void);
    CHAR _far *nameptr;
    CHAR name [9];

    name [8] = '\0';

    printf (header);

    /*   Trace the handler chain to its end, reporting the address of each
         handler and the name of its owner. */

    callpoint = lpXMS;
    do {
        printf ("\n %Fp      ", callpoint);

        nameptr = GetMemoryOwner ((void _far *) callpoint);
        if (nameptr) {
            _fmemcpy (name, nameptr, 8);
            printf ("%-8s", name);
        }
        else printf ("        ");

    } while (callpoint = FollowXMSChain (callpoint));
}
```

Identification of handlers by program name is performed by calling routines in the ARENA.C module of Chapter 4. The module will be used again by the resident part of the full XMSWATCH utility.

```
/* memowner.c */

#include    <arena.h>
#include    <psp.h>
#include    <standard.h>

extern _segment GetArenaSegment (_segment);                         // arena.c
extern _segment GetSubArenaSegment (_segment, _segment); // arena.c
extern BYTE GetValidArenaID (_segment);                             // arena.c
extern BYTE GetValidSubArenaID (_segment);                          // arena.c

#ifdef _MANUAL_UMBLINK_
extern struct {
    BYTE minor;
    BYTE major;
} wDOSVersion;
#define _osmajor wDOSVersion.major
#define _osminor wDOSVersion.minor
#else
extern BYTE _osminor;                           // defined in STDLIB.H
extern BYTE _osmajor;                           // defined in STDLIB.H
#endif

#define MK_ARENA_FP(seg)    ((struct ARENA_HEADER _far *) MK_FP (seg, NULL))
#define MK_PSP_FP(seg)          ((struct PSP _far *) MK_FP (seg, NULL))

CHAR _far * _far GetMemoryOwner (void _far *address)
```

```
{
    register _segment testseg, ownerseg;
    _segment blockseg;
    WORD temp;
    BYTE id;

    /*   Determine which paragraph contains the given address, checking
         that the paragraph calculation does not overflow 16 bits. */

    temp = (WORD) address >> 4;
    if ((_segment) address > 0xFFFF - temp) return (NULL);
    testseg = (_segment) address + temp;

    /*   Find the memory arena header for the block containing the desired
         segment. */

    blockseg = GetArenaSegment (testseg);
    if (blockseg == 0) return (NULL);

    /*   If it seems plausible that the block's owner is indeed the PSP that
         obtained the allocation from DOS, then return a pointer to the name
         field in the arena header belonging to the PSP. This demonstration
         makes no attempt (under early DOS versions) to retrieve program
         names from environment blocks. */

    ownerseg = MK_ARENA_FP (blockseg) -> spOwner;
    if (GetValidArenaID (ownerseg - 1) != 0) {
         if (MK_ARENA_FP (ownerseg - 1) -> spOwner == ownerseg) {
              if (_osmajor < 4) return (NULL);
              else return (MK_ARENA_FP (ownerseg - 1) -> cName);
         }
    }
    /*   Otherwise, the block is not owned by a program. Not much can be
         done in this case without the subarena system introduced with DOS 4
         for identifying device drivers (and IFSs) by their filenames. */

    if (_osmajor < 4) return (NULL);

    blockseg = GetSubArenaSegment (blockseg, testseg);
    if (blockseg == 0) return (NULL);
    id = GetValidSubArenaID (blockseg);
    if (id == 'D' OR id == 'I') return (MK_ARENA_FP (blockseg) -> cName);

    return (NULL);
}
```

A Front End for XMS Extension

The main problem for developers of a program that hooks XMS services is that the structure defined for the header is not easily produced from high-level languages, most of which impose some sort of prologue code. However, it is fairly easy to reach back to Chapter 7, where an assembly-language front-end was developed to reconcile C to the particular difficulties of the device-driver format, and snag that code for use as an assembly-language front-end for XMS extension. The following front-end looks strikingly similar to DRIVER.ASM in Chapter 7. Naturally, XMSWATCH will use this XMS-extension facility to spy on calls that other programs make to the XMS allocation and free functions.

```
; xms.asm

;   This file provides a C-language interface for intercepting XMS calls.

.NOLIST
  INCLUDE         intpack.inc
  INCLUDE         keepsegs.inc
.LIST

.MODEL                        SMALL, C, FARSTACK

;   It is assumed that the presence of a machine supporting extended memory
;   will have been established before enabling this module and, consequently,
;   that the PUSHA and POPA instructions are available.

.286

;   ======================================================================

.FARDATA            KEEP

;   XMS handlers are installed by threading themselves into a chain without
;   disturbing the original entry point. This ensures that the most recent
;   addition has first option on all XMS requests, even those issued by
;   programs activated earlier.

;   The address at which to call the next handler in the chain must be
;   provided externally.

                              EXTERN    lpNextXMS:DWORD

;   Jump table for XMS filters:
```

```
                             PUBLIC   XMSFilterTable
XMSFilterTable               LABEL    WORD

;   All procedures in the table have C language prototypes of the form

;         unsigned XMSFilter (struct FARPACK STACKSEG *);

;   Functions not provided in other modules default to one that is
;   transparent.

NEXTNUM            =         00h
@XMSFilter                   MACRO    num:REQ, command:REQ
  IF                         num LT NEXTNUM
    .ERR           <@XMSFilter macro not in sequence>
  ENDIF
  IF                         num GT NEXTNUM
    REPEAT                   num - NEXTNUM
                             dw       OFFSET KEEP_TEXT:XMSFilterPass
    ENDM
  ENDIF
  .CODE            KEEP_TEXT
                             EXTERN   XMSFilter&command (XMSFilterPass):NEAR
  .FARDATA                   KEEP
                             dw       OFFSET KEEP_TEXT:XMSFilter&command
  NEXTNUM          =         num + 01h
ENDM

@XMSFilter                   00h, GetVersion
@XMSFilter                   01h, AllocateHMA
@XMSFilter                   02h, FreeHMA
@XMSFilter                   03h, GlobalEnableA20
@XMSFilter                   04h, GlobalDisableA20
@XMSFilter                   05h, LocalEnableA20
@XMSFilter                   06h, LocalDisableA20
@XMSFilter                   07h, QueryA20
@XMSFilter                   08h, QueryFreeMemory
@XMSFilter                   09h, AllocateEMB
@XMSFilter                   0Ah, FreeEMB
@XMSFilter                   0Bh, MoveData
@XMSFilter                   0Ch, LockEMB
@XMSFilter                   0Dh, UnlockEMB
@XMSFilter                   0Eh, GetHandleInfo
@XMSFilter                   0Fh, ResizeEMB
@XMSFilter                   10h, AllocateUMB
```

```
@XMSFilter                  11h, FreeUMB

;   A private variable to support stack restitution by the XMSFilterPass
;   function

pStack                      dw      ?

;   ======================================================================

.CODE                       KEEP_TEXT

xms                         PROC    FAR PUBLIC

;   XMS handlers begin with the following instruction sequence, which may be
;   replaced by an intersegment jump should another handler need to be
;   installed.

                            jmp     short   @f
                            nop
                            nop
                            nop
@@:

;   Save the caller's registers on the stack. As with _interrupt functions
;   (for which the same structure is used), this ensures the preservation of
;   registers across the XMS call unless deliberate action. is taken To support the
;   return of different values in the registers, the XMSFilter functions are
;   all passed a stack-based pointer to the register copy.

                            pusha
                            push    ds
                            push    es
                            mov     bp,sp

;   Although the CPU flags are not included in the structure of stacked
;   registers, they must also be saved across the XMS call.

                            pushf

;   Load ds with the segment address of the resident data group.

                            mov     dx,KEEP_GROUP
                            mov     ds,dx
                            ASSUME  ds:KEEP_GROUP
```

```
;   Save the stack pointer to allow the XMSFilterPass function to recover
;   the entry conditions.

                            push      [pStack]           ; for re-entrancy
                            mov       [pStack],sp

;   Safeguard against incomplete initialization. This supports more
;   generality in the routine that patches in the far jump when hooking the
;   XMS. Without this provision, that routine would need to have knowledge
;   of the lpNextXMS variable's address to initialize it before the patch.
;   Thus, in the short time between hooking into the XMS chain and setting
;   lpNextXMS (to reconnect the chain), return the "unsupported function"
;   error for all XMS calls in preference to having XMS requests go astray.

                            mov       dx,word ptr [lpNextXMS]
                            or        dx,word ptr [lpNextXMS + 02h]
                            jnz       @f

                            mov       (REGPACK PTR [bp]).stacked_ax,0000h
                            mov       (REGPACK PTR [bp]).stacked_bl,80h

                            jmp       done
@@:

;   Determine which procedure should handle the XMS function.

                            mov       al,ah
                            cmp       al,NEXTNUM
                            jnb       too_high

                            xor       ah,ah
                            mov       si,ax
                            shl       si,1
                            mov       si,[si + XMSFilterTable]

                            jmp       @f
too_high:
                            mov       si,OFFSET XMSFilterPass
@@:

;   Pass each filter a pointer to the caller's registers saved on the stack.
;   The register values passed directly to the filter are all undefined,
;   except that ds = KEEP_GROUP.

                            push      bp
```

```
                        call    si
                        pop     bx
done:

;   By returning normally, a filter indicates that it has completed the XMS
;   request. Register values for the XMS caller will have been entered into
;   the image on the stack, so retrieve them and return.

                        pop     [pStack]
                        popf

                        pop     es
                        pop     ds
                        popa

                        ret

xms                     ENDP

;   ---------------------------------------------------------------------

XMSFilterPass           PROC    PUBLIC

;   At any time, a filter may pass the XMS call to the next handler by
;   calling this function. This is much the same as using the _chain_intr function
;   in the Microsoft C library, except that in this case, the destination
;   for the control transfer is fixed and need not be passed as an argument.

;       void XMSFilterPass (void);

                        mov     sp,[pStack]
                        pop     [pStack]
                        popf

                        pop     es
                        pop     ds
                        popa

;   The address of the next handler is kept in KEEP_GROUP, which is
;   convenient for the C-language interface but makes an indirect jump
;   impossible without corrupting the XMS caller's segment registers. Place
;   the address on the stack in such a way that a RETF transfers control
;   correctly.

                        push    ax          ; subtract four bytes from stack
```

```
                             push     ax        ; without disturbing the flags

                             push     bp
                             mov      bp,sp
                             push     ds
                             push     ax
                             mov      ax,KEEP_GROUP
                             mov      ds,ax
                             lds      ax,[lpNextXMS]
                             mov      [bp + 02h],ax ; place the transfer address
                             mov      [bp + 04h],ds ; over the four extra bytes
                             pop      ax
                             pop      ds
                             pop      bp

                             retf

XMSFilterPass                ENDP

;   ------------------------------------------------------------------------

;   A function is provided for calling the next XMS handler in the chain.
;   Registers are taken from a structure on the stack (which need not be the
;   one in which the caller's registers were saved).

;        unsigned CallNextXMS (struct REGPACK STACKSEG *);

CallNextXMS                  PROC     PUBLIC USES ds di si, regptr:WORD
                             LOCAL    handler:DWORD

                             les      bx,[lpNextXMS]

                             mov      word ptr [handler],bx
                             mov      word ptr [handler + 02h],es

;   Load registers with values for the XMS server. In version 2.0 of the
;   specification, only registers ax, bx, dx, si, and ds are ever meaningful.

                             push     bp
                             mov      bp,[regptr]
                             ASSUME   bp:PTR INTPACK

                             mov    ds,[bp].stacked_ds
                             mov    si,[bp].stacked_si
                             mov      dx,[bp].stacked_dx
```

```
                              mov       bx,[bp].stacked_bx
                              mov       ax,[bp].stacked_ax

                              pop       bp
                              ASSUME    bp:NOTHING

;   Call the required XMS handler.

                              call      [handler]

;   Copy the return values onto the stack. XMS version 2.0 returns values
;   in registers ax, bx, and dx only. The implementation in HIMEM.SYS
;   preserves cx, si, di, ds, and es explicitly. It is also capable of
;   corrupting values in bx and dx even if the specification assigns them
;   no meaning.

                              push      bp
                              mov       bp,[regptr]
                              ASSUME    bp:PTR INTPACK

                              mov       [bp].stacked_ax,ax
                              mov       [bp].stacked_bx,bx
                              mov       [bp].stacked_dx,dx

                              pop       bp
                              ASSUME    bp:NOTHING

                              ret

CallNextXMS                   ENDP

END
```

Spying on XMS Handle Allocations

Now to put it all to good use! XMSWATCH reports extended memory usage. The program may be run as a normal DOS executable or loaded with a DEVICE statement in CONFIG.SYS. It is recommended that it be loaded as a device driver with the /INSTALL option immediately after the main XMS server. /INSTALL retains a copy in memory to monitor XMS activity. Without this switch, the program cannot describe extended memory that has been already allocated.

First, the code that monitors the XMS calls to allocate and release extended memory; of course, the following MONITOR.C module uses the XMS extension facility provided by XMS.ASM.

You may wonder why XMSWATCH does not also monitor XMS realloc calls. The reason is that the resident component of XMSWATCH merely wants to know the allocated handles; it doesn't care what block sizes clients ask for. Later, the non-resident component will use the handles to query block sizes, lock counts, and so on. Note this need to independently check the size of each allocated EMB. Under HIMEM, at least, it is possible for function 08h to allocate a larger block than requested, and the client would never know.

```
/* monitor.c - this is a resident module. */

#include    <dos.h>
#include    <memory.h>

#include    <intpack.h>
#include    <standard.h>
#include    <sysvars.h>

#include    "xmswatch.h"

#pragma intrinsic (_disable, _enable, memset)

/*  The XMS entry point at the time of XMSWATCH's installation  */
void (_far *lpOrgXMS) (void) = 0;

/*  The address at which XMS requests would have been processed (after the
    chain of far jumps) were this module not intercepting them     */
void (_far *lpNextXMS) (void) = 0;

/*  Functions that either jump to or call lpNext, but first load register
    values from the structure passed on the stack  */
extern void XMSFilterPass (void);
extern int CallNextXMS (struct REGPACK STACKSEG *);

/*  Items to help identify the DOS program to which a given address belongs  */
extern CHAR _far * _far GetMemoryOwner (void _far *);
struct SYSVARS _far *lpSysvars = 0;

typedef struct {            // a ruse to treat an eight-character
    CHAR progname [8];       // program name as a single object
} PROGNAME;                     // for more efficient manipulation

/*  Variables that describe the table in which information about XMS
    allocations is kept  */
WORD wHandles = 0;
struct HANDLE *pHandleTable = 0;
```

```
WORD wHandleTableSize = 0;

/*  ========================================================================  */

/*  The handle table is created in memory that was originally used by the
    program's initialization code. The Activate function is called from a
    safe location just before the program terminates and stays resident.
    Until the table has been cleared and the bActive flag is set to TRUE,
    all XMS functions are passed transparently to the next handler. */

static BOOL bActive = FALSE;

void _far Activate (void)
{
    memset (pHandleTable, 0, wHandleTableSize);
    bActive = TRUE;
}

/*  The following function searches the handle table for the entry that
    matches the handle supplied as the function's argument. It returns the
    address of the corresponding HANDLE structure or NULL if the handle is
    not yet known to the table. */

static struct HANDLE *GetHandleDescriptor (WORD handle)
{
    register struct HANDLE *hptr;
    register WORD n;

    n = wHandles, hptr = pHandleTable;
    if (n) {
         do {
              if (hptr -> wHandle == handle) return (hptr);
         } while (++ hptr, -- n);
    }
    return (NULL);
}

/*  ------------------------------------------------------------------------  */

/*  Only two XMS functions are intercepted non-trivially—the ones for
    allocating and releasing an extended memory block. The objective is to
    record the numerical value of the handle returned by the XMS server so
    that the non-resident component of this program may obtain information
    about the allocation later. In addition, some attempt is made to
    identify the DOS program requesting the allocation. */
```

```
void XMSFilterAllocateEMB (register struct FARPACK STACKSEG *regptr)
{
    register struct HANDLE *hptr;
    CHAR _far *nameptr;

    /*   Do not intercept the function if the handle table has not yet been
         initialized. */

    if (NOT bActive) XMSFilterPass ();

    /*   XMS functions must be reentrant. For present purposes, dodge the
         problem by preventing interrupts. Since allocation requests are
         infrequent, this is not intolerable. */

    _disable ();

    /*   Call the next handler to deal with the request as the caller
         intends, but examine the result before allowing control to return. */

    if (CallNextXMS ((struct REGPACK STACKSEG *) regptr)) {

        /* If the call was successful, find a spare entry in the handle
           table and record the handle. */

        hptr = GetHandleDescriptor (0);
        if (hptr != NULL) {
            hptr -> wHandle = regptr -> dx;

            /*   Try also to identify the DOS program that sought the XMS
                 allocation, copying the program name (eight characters) from
                 the DOS memory arena header if this can be located. */

            if (nameptr = GetMemoryOwner (regptr -> retaddr))
                 *((PROGNAME *) hptr->cOwner) = *((PROGNAME _far *)nameptr);
        }
    }
}

void XMSFilterFreeEMB (struct FARPACK STACKSEG *regptr)
{
    register WORD handle;
    register struct HANDLE *hptr;

    if (NOT bActive) XMSFilterPass ();
    handle = regptr -> dx;

    _disable ();
    if (CallNextXMS ((struct REGPACK STACKSEG *) regptr)) {

         /*  If a handle in the table has been released, clear the
             corresponding entry before returning to the XMS caller. */

         hptr = GetHandleDescriptor (handle);
         if (hptr != NULL) memset (hptr, 0, sizeof (*hptr));
    }
}
```

Next is the non-resident code that installs the monitor, using the `ThreadXMSChain` function from the earlier XMSCHAIN.C module.

```
/* install.c */

#include     <dos.h>
#include     <stdio.h>
#include     <stdlib.h>

#include     <driver.h>
#include     <keepc.h>
#include     <psp.h>
#include     <standard.h>
#include     <sysvars.h>

#include     "xmswatch.h"

#pragma intrinsic (_disable, _enable)

/*  A flag provided by the CRTDRVR library to distinguish normal program
    execution from device driver initialization  */
extern BOOL _RESIDENT bLoadedAsDriver;

/*  The logical name for access to the resident component's device driver
    interface  */
CHAR DeviceName [] = "XMSWWWW0";

/*  The calling point for XMS services, as determined by MAIN.C  */
extern void (_far *lpXMS) (void);
```

```
/*  Calling address needed by the resident component  */
extern void (_far * _RESIDENT lpNextXMS) (void);

/*  The address of the handler (in MONITOR.C) that this program adds to
    the XMS chain, a function to perform the addition and a resident
    preterminator to activate the handler */

extern void _far xms (void);
extern void (_far *ThreadXMSChain (void (_far *) (void))) (void);
extern void _far Activate (void);

/*  The address of DOS's internal variable table (recorded in the resident
    data group by MAIN.C) and of this program's own device driver header, so
    that device driver installation can be mimicked even if the program is
    run as a normal executable */

extern struct SYSVARS _far * _RESIDENT lpSysvars;
extern struct DRIVER_HEADER _far DriverHeader;

/*  A function to obtain the number of XMS handles to watch, plus variables
    to describe the tables in which information about these handles will be
    kept */

extern WORD QueryFreeHandles (void);

extern WORD _RESIDENT wHandles;
extern struct HANDLE * _RESIDENT pHandleTable;
extern WORD _RESIDENT wHandleTableSize;

/*  A self-explanatory function for fatal message output  */
extern PutError (CHAR *, ...);

/*  ====================================================================  */

int Install (void)
{
    struct find_t fileinfo;
    WORD paras_avail, paras_wanted;

    /*  Disallow an attempt at multiple installation, indicated by the
        prior existence of the XMSWWWW0 device. Be cautious and do not
        proceed if a file of the same name exists, because installation would
        render that file inaccessible for the remainder of the session. */

    if (_dos_findfirst (DeviceName,
          _A_HIDDEN | _A_SYSTEM | _A_SUBDIR, &fileinfo) == 0) {
      PutError ("XMSWWWW0 file or device present already");
      return (-1);
}

/*    Determine (and save for the resident component) the number of free
      XMS handles remaining. It is not possible to obtain via the XMS the
      number of handles that can be provided by the server, so if XMSWATCH
      is not loaded immediately after the primary XMS server, some handles
      may be missed. */

wHandles = QueryFreeHandles ();
if (wHandles == 0) {
      PutError ("no XMS handles available");
      return (-1);
}

/*    Check the availability of space for the table in which the resident
      component will maintain information about XMS handles. The handle
      table begins at the break address (NOKEEP_BTEXT:0000h) specified by
      the Keep C library. */

wHandleTableSize = wHandles * sizeof (struct HANDLE);
paras_wanted = (wHandleTableSize + 0x0F) >> 4;
paras_avail = ((struct PSP _far *) MK_FP (_psp, NULL)) -> spCeiling
                         - (_segment) lpBreakAddress;

if (paras_wanted > paras_avail) {
      PutError ("insufficient memory for handle table");
      return (-1);
}

/*    Calculate and record the offset address of the handle table with
      respect to the resident data group (the segment address of which is
      held in spKEEP). */

pHandleTable = (struct HANDLE *)
                   ((WORD) ((_segment) lpBreakAddress - spKEEP) << 4);

/*    Thread this program's own XMS handler onto the end of the chain. */

lpNextXMS = ThreadXMSChain (xms);
if (lpNextXMS == NULL) {
      PutError ("bad XMS handler chain");
      return (-1);
```

```
    }

    /*  Note that the interception of XMS calls cannot be activated until
        the handle table has been initialized (cleared to zero), but the
        handle table lies in space presently occupied by nonresident code.
        Register the activation routine as a resident preterminator to be
        sure it will be called from a safe location. */

    AtExitKeepC (Activate);

    /*  If the program has not been loaded as a device driver, thread it
        into the device-driver chain by hand. Do not take the chance
        (however small) that an interrupting process will try to follow the
        device-driver chain (by hand or through a DOS function) while the
        chain is being adjusted. */

    if (NOT bLoadedAsDriver) {
        DriverHeader.lpNext = lpSysvars -> NULHeader.lpNext;
        _disable ();
        lpSysvars -> NULHeader.lpNext = &DriverHeader;
        _enable ();
    }

    /*  Display an installation message. */

    printf ("\nXMSWATCH driver installed - monitoring %d handles\n", wHandles);

    /*  Tidy up, as usual. Retain enough memory to cover the handle table. */

    FreeEnvironment ();
    CloseAllFiles ();
    ExitKeepC (0, (BYTE _RESIDENT *) pHandleTable + wHandleTableSize);
}
```

The next module is API.C, which is the resident portion of XMSWATCH used to return information about XMS allocations to the non-resident portion. API.C provides a device-driver interface, which another part of XMSWATCH (or, for that matter, another program) can query by making IOCTL Read calls to the XMSWWWW0 device.

```
/* api.c */

#include    <driver.h>
#include    <standard.h>

#include    "xmswatch.h"
```

```
extern WORD wHandles, wHandleTableSize;
extern struct HANDLE *pHandleTable;

extern void (_far *lpNextXMS) (void);

static void CopyWatchData (struct XMSWATCH _far *);
static void CopyHandleTable (BYTE _far *);

/*  ==========================================================================  */

/*  Provide information as control data for the XMSWWWW0 device. At DOS
    function level, this means supporting the IOCTL Read function (4402h)
    for which the caller indicates the number of bytes it expects the device
    to copy to a given buffer. The caller should first ask for the XMSWATCH
    structure (whose size is fixed). One of that structure's fields is the
    size of the handle table (which depends on the number of XMS handles
    available when XMSWATCH is installed). A second IOCTL Read should then
    return the whole handle table. */

WORD DrvrProcIOCTLRead (struct REQUEST_HEADER _far *lpReqHdr)
{
    if (lpReqHdr -> wCount) {
          if (lpReqHdr -> wCount == sizeof (struct XMSWATCH))
              CopyWatchData ((struct XMSWATCH _far *) lpReqHdr -> lpBuffer);
          else if (lpReqHdr -> wCount == wHandleTableSize)
              CopyHandleTable (lpReqHdr -> lpBuffer);
          else lpReqHdr -> wCount = 0;
    }
    return (D_DONE);
}

static void CopyWatchData (struct XMSWATCH _far *ptr)
{
    ptr -> wVersion = XMSWATCH_VERSION;
    ptr -> wHandles = wHandles;
    ptr -> wHandleTableSize = wHandleTableSize;
    ptr -> lpNextXMS = lpNextXMS;
}

static void CopyHandleTable (BYTE _far *ptr)
{
    register BYTE *src = (BYTE *) pHandleTable;
    register BYTE _based ((_segment) ptr) *dest = ptr;
    WORD n = wHandleTableSize;
    do {
          *dest ++ = *src ++;
    } while (-- n);
}
```

CONSULT.C is the non-resident portion of XMSWATCH which retrieves information from the resident portion; as already noted, it issues IOCTL Read calls to the XMSWWWW0 device.

```
/* consult.c */

#include    <dos.h>
#include    <fcntl.h>

#include    <standard.h>

#include    "xmswatch.h"

/*  Functions and data used to detect the presence of a resident copy of
    XMSWATCH and to retrieve the information it may have saved      */

extern CHAR DeviceName [8];

WORD _dos_getdevicestatus (int, WORD *);
WORD _dos_ioctlread (int, void _far *, WORD, WORD *);

/*  The following function obtains two data structures from the resident
    XMSWATCH copy. The function's caller need not know that the data is
    obtained via a device driver IOCTL interface but must understand the
    form in which the data is presented. The XMSWATCH structure has a fixed
    size and the caller must supply the address of some suitable memory.
    Since the LIST structure contains a field of variable length, the caller
    cannot know in advance how much memory to set aside. Instead, the
    function retrieves the list into memory allocated from the heap and
    returns the address. */

struct LIST *ConsultWatcher (register struct XMSWATCH *watchptr)
{
    int devicehandle;
    register WORD handle;
    WORD count, status;
    struct LIST *listptr;

    /*   Attempt to open the XMSWWWW0 device. */
    if (_dos_open (DeviceName, O_RDWR, &devicehandle)) return (FALSE);
    /*   Use DOS function 4400h to verify that _dos_open has indeed found a
         character device (status flags masked by 0x0080) that supports the
         IOCTL interface (mask 0x4000). */
    if (_dos_getdevicestatus (handle = devicehandle, &status)
            OR (status & 0x4080) != 0x4080) goto fail;

    /*   Read the XMSWATCH structure and check that the resident XMSWATCH
         copy supports the expected version of the interface. */
    if (_dos_ioctlread (handle, watchptr, sizeof (*watchptr), &count)
            OR count != sizeof (*watchptr)) goto fail;

    if (watchptr -> wVersion != XMSWATCH_VERSION) goto fail;

    /*   Information in the MONITOR structure yields the size of the resident
         handle table. Ask the device to copy the table into space allocated
         from the near heap. */

    listptr = CreateList (watchptr -> wHandles);
    if (listptr == NULL) goto fail;

    if (_dos_ioctlread (handle, listptr -> array,
            watchptr -> wHandleTableSize, &count)
            OR count != watchptr -> wHandleTableSize) goto fail;

    /*   Close the handle and return a pointer to the dynamically-allocated
         list. */
    _dos_close (handle);
    return (listptr);

    fail:

    /*   If, for any reason, the device is unable to return the expected data,
         close the file handle and return NULL. The goto statement has been
         used to deal with this cleanup because the alternative involves
         either too many braces and indents or too much repetition. */

    _dos_close (handle);
    return (NULL);
}

WORD _dos_getdevicestatus (int handle, WORD *statusptr)
{
    WORD status;
    _asm {
            mov     bx,handle
```

```
                mov     ax,4400h
                int     21h
                jc      fail
                mov     [status],dx
    }
    *statusptr = status;
    return (0);

    _asm fail:
}

WORD _dos_ioctlread (int handle, void _far *buffer, WORD count, WORD *numread)
{
    _asm {
                push    ds
                lds     dx,buffer
                mov     cx,count
                mov     bx,handle
                mov     ax,4402h
                int     21h
                pop     ds
                jc      fail
                mov     [count],ax
    }
    *numread = count;
    return (0);

    _asm fail:
}
```

WATCHED.C, which follows, uses the functions from LIST.C in Chapter 11 to build a list of structures describing the EMBs monitored by the resident XMSWATCH component.

```
/* watched.c */

#include    <stdio.h>

#include    <standard.h>

#include    "xmsfunc.h"
#include    "xmswatch.h"

/*  XMS services  */

extern void (_far *lpXMS) (void);
extern BOOL LockEMB (WORD, DWORD *);
extern BOOL GetEMBInfo (WORD, struct EMBINFO *);

/*  Two variables and one function for retrieving information from a
    resident copy of XMSWATCH  */

static struct LIST *pWatchedEMBs;
static struct XMSWATCH xmswatch;

extern struct LIST *ConsultWatcher (struct XMSWATCH *);

/*  Results are presented in a multi-column table under the following
    header. Data for each column is handled by a separate function that
    pads the item to the expected width. Adjacent columns are separated
    by five spaces. */

static CHAR banner [] = "\n\
Extended Memory Allocation Detected by Resident Component of XMSWATCH:";

static CHAR header [] = "\n\
\n\
  Linear Address  Size (KB)  Locks   Handle     Owner         Properties \n\
  ----------------------------------------------------------------------";

static void PutLocks (struct HANDLE *);
static void PutHandle (struct HANDLE *);
static void PutOwner (struct HANDLE *);
static void PutProperties (struct HANDLE *);

static void SpaceOut (void);

/*  A few functions developed for FREE.C help harmonize this report with the
    one about free memory blocks. */

extern void PutLinearAddress (struct HANDLE *);
extern void PutSizeInK (struct HANDLE *);
extern void PutShortfall (WORD);

extern WORD BuildListOfFreeEMBs (struct LIST *);

/*  Material needed for comparison of this report with the one that FREE.C
    generates  */

extern struct LIST *pFreeEMBs;
```

```
static int TestRegion (DWORD, WORD);

#define NONE_FREE   0
#define SOME_FREE   1
#define ALL_FREE    2

static BOOL IsAvailableHandle (WORD);

/*  ======================================================================  */

void ReportWatchedEMBs (void)
{
    register struct HANDLE *ptr;
    register WORD handle;
    WORD unaccounted;
    void (_far *xmsptr_save) (void);
    struct EMBINFO info;

    /*  Information about EMBs that have been allocated already can be
        obtained only by intercepting the XMS calls at the time of
        allocation to record the corresponding handle. If the resident
        component of the program (the watcher) has been installed, retrieve
        its data. */

    pWatchedEMBs = ConsultWatcher (&xmswatch);
    if (pWatchedEMBs == NULL) return;

    /*  One essential piece of data is the XMS calling address understood by
        the earlier XMSWATCH installation. This is returned as one of the
        fields in the XMSWATCH structure and must replace lpXMS (the present
        XMS calling point) temporarily, so that questions about monitored
        handles can be directed to the XMS server that actually allocated
        the handles.

        Observe that the lpNextXMS field is the destination of the far jump
        that started the XMS handler hooked by the watcher. The following
        code therefore undercuts subsequent additions to the XMS chain—it
        is technically illegal and it is essential that operations conducted
        this way be noninvasive. Really, the onus is on the XMS monitoring
        code in MONITOR.C to watch more functions than the ones implemented
        for this demonstration version of the program. */

    xmsptr_save = lpXMS;
    lpXMS = xmswatch.lpNextXMS;
```

```
/*   Each entry in the resident handle table (a copy of which is now at
     the address pointed to by pWatchedEMBs) contains just a handle and a
     string describing the DOS program that obtained the allocation. For
     each handle monitored this way, obtain the base address, size, and
     lock count of the EMB it denotes, then sort the entry in order of
     increasing base address. */

for (ptr = pWatchedEMBs -> pUnused; ptr < pWatchedEMBs -> pEnd; ptr ++) {
     if (handle = ptr -> wHandle) {
          if (GetEMBInfo (handle, &info)) {
                ptr -> cLocks = info.cLocks;
                ptr -> wSizeInK = info.wSizeInK;
                ptr -> wFlags = USED_EMB;
                if (LockEMB (handle, &(ptr -> dwBase))) {
                     ptr -> wFlags |= PHYSICAL_EMB;
                     UnlockEMB (handle);
                }
                LinkEntry (pWatchedEMBs, ptr);
          }
     }
}

/*   Add information about the free memory blocks as understood by the
     XMS server whose activity has been monitored, then restore lpXMS to
     the current XMS calling point. */

unaccounted = BuildListOfFreeEMBs (pWatchedEMBs);

lpXMS = xmsptr_save;

/*   Display the results only if at least one nontrivial EMB (whether
     allocated or still free) has been discovered. In the case where no
     block can be found (most likely because all extended memory was
     allocated before the watcher's installation), the list may be useful
     to a future extension of the program because it provides numerical
     values for the XMS server's spare handles. */

ptr = pWatchedEMBs -> pLowest;
for (;;) {
     if (ptr == NULL) return;
     if (ptr -> wFlags & (USED_EMB | FREE_EMB)) break;
     ptr = ptr -> pNext;
}

printf (banner);
```

```
    printf ("\n\n  Handles monitored:  %u", xmswatch.wHandles);
    printf (header);

    /*   Describe the blocks in order of ascending linear address. */

    for (ptr = pWatchedEMBs -> pLowest; ptr != NULL; ptr = ptr -> pNext) {
         if (ptr -> wFlags & (USED_EMB | FREE_EMB)) {
             printf ("\n     ");
             PutLinearAddress (ptr); SpaceOut ();
             PutSizeInK (ptr); SpaceOut ();
             PutLocks (ptr); SpaceOut ();
             PutHandle (ptr); SpaceOut ();
             PutOwner (ptr); SpaceOut ();
             PutProperties (ptr);
         }
    }
    if (unaccounted) PutShortfall (unaccounted);
    printf ("\n");
}

/*  The Locks, Handle, and Owner headings are relevant only for allocated
    EMBs. For free blocks, maintain column alignment by padding the display
    with an equivalent number of spaces. */

static void PutLocks (struct HANDLE *ptr)
{
    if (ptr -> wFlags & USED_EMB) printf ("%3u", ptr -> cLocks);
    else printf ("   ");
}

static void PutHandle (struct HANDLE *ptr)
{
    if (ptr -> wFlags & USED_EMB) printf ("%04Xh", ptr -> wHandle);
    else printf ("     ");
}

static void PutOwner (struct HANDLE *ptr)
{
    if (ptr -> wFlags & USED_EMB) printf ("%-8s", ptr -> cOwner);
    else printf ("        ");
}

static void SpaceOut (void)
{
    printf ("     ");
```

```
}

/*  The most straightforward purpose of the Properties heading is to avoid
    interfering with other columns when indicating that a block is free.
    This is only a question of taste—free blocks are
    distinguished by having no handle or owner (some allocated blocks may
    appear to have no owner because the XMS caller outside the DOS
    memory arena system at the time of allocation).

    A greater purpose is to compare the extended memory picture seen by the
    XMS server whose activity has been monitored by the resident XMSWATCH
    copy with the picture currently presented to applications. Extensions
    to the XMS chain can produce very curious effects. It is the job of the
    PutProperties function to report possible discrepancies. */

static void PutProperties (register struct HANDLE *ptr)
{
    DWORD linear_address;
    struct EMBINFO info;

    /*    If the block described by ptr is free, say so. */

    if (ptr -> wFlags & FREE_EMB) {
         printf ("free");

         /*  Note that although the XMS server monitored by the
             resident XMSWATCH copy regards the block as free, a subsequent
             extension to the XMS chain might not be making the block
             available. */

         if (ptr -> wFlags & PHYSICAL_EMB) {
             if (TestRegion (ptr -> dwBase, ptr -> wSizeInK) != ALL_FREE)
                  printf (" but hidden");
         }
         return;
    }

    /*   A block of memory that has been allocated (but not yet released) at
         the level monitored by the resident copy of XMSWATCH might seem free
         to a program making XMS calls now. A likely cause is that a program
         is redistributing memory to its clients, having hooked the XMS chain
         to control its clients' use of extended memory. The condition can
         be detected only if some piece of the original allocation remains
         undistributed. */
```

```
if (ptr -> wFlags & PHYSICAL_EMB) {
     if (ptr -> wSizeInK != 0) {
         if (TestRegion (ptr -> dwBase, ptr -> wSizeInK) != NONE_FREE) {
              printf ("sub-let");
              return;
         }
     }
}

/*   An XMS extension might also be redistributing handles obtained from
     the original XMS server. This will be revealed if a handle already
     assigned to an EMB (and detected by the resident XMSWATCH copy) also
     turns up in the pFreeEMBs list (as one of the current XMS server's
     spares). */

if (IsAvailableHandle (ptr -> wHandle)) {
     printf ("consumed");
     return;
}

/*   Other discrepancies may be uncovered by asking legally for
     information about each handle, that is, by issuing XMS calls to the whole
     XMS chain rather than by undercutting to the handler monitored by the
     resident copy of XMSWATCH. Extensions to the chain might not
     recognize a handle allocated before their addition or might
     associate the same handle with a different block of memory.

     Be aware though, that an XMS handle is normally a private matter
     between the XMS server and the client that obtains the allocation.
     XMSWATCH interferes with this relationship by eavesdropping on XMS
     activity. A program that hooks the XMS chain after obtaining a
     handle to an EMB might react unfavorably to seeing that handle
     appear in an XMS request issued by another program. This is the
     case with the enhanced mode of Windows 3.0, which deems this
     activity to violate system integrity. The consequences are
     harmless, but if the compatibility is important, the following
     conditional compilation block should be skipped. */

#if TRUE

if (NOT GetEMBInfo (ptr -> wHandle, &info)) {
     printf ("invalidated");
     return;
}
if (ptr -> wFlags & PHYSICAL_EMB) {
```

```
         if (NOT LockEMB (ptr -> wHandle, &linear_address)) {
             printf ("virtualised");
             return;
         }
         UnlockEMB (ptr -> wHandle);
         if (ptr -> dwBase != linear_address
                   OR ptr -> wSizeInK != info.wSizeInK) {
             printf ("converted");
             return;
         }
    }
    #endif
}

/*  Given a region of extended memory, the next function assesses whether
    all, any, or none of the region is currently available through the XMS.
    The function relies on the pFreeEMBs list built by code in the FREE.C
    module. */

static int TestRegion (DWORD test_base, WORD test_sizeink)
{
    register struct HANDLE *ptr;
    DWORD ptr_top, test_top = test_base + ((DWORD) test_sizeink << 10);
    int result = NONE_FREE;

    /*   Search the list that describes the blocks and handles seen at
         application level. The list is arranged in increasing order of base
         address, so abort the search on finding the first block that lies
         entirely above the test region. */

    ptr = pFreeEMBs -> pLowest;
    while (ptr != NULL AND ptr -> dwBase < test_top) {

         /*  Entries corresponding to spare handles are of no interest here
             and neither are blocks of virtual memory for which a base
             address could not be determined. Some overlap between the
             pFreeEMBs entry under consideration and the test region is also
             required. */

         if (ptr -> wFlags & FREE_EMB AND ptr -> wFlags & PHYSICAL_EMB) {
             ptr_top = ptr -> dwBase + ((DWORD) ptr -> wSizeInK << 10);
             if (ptr_top > test_base) {

                   /*  The simple case arises when the block described by ptr
                       contains the test region. */
```

```
                if ( ptr -> dwBase <= test_base AND ptr_top >= test_top)
                    return (ALL_FREE);

                /*  Otherwise, the possibility remains that the test region
                    is the union of two or more blocks in the pFreeEMBs
                    list, so continue the algorithm for that part of the
                    test region that has not yet been covered. */

                test_base = ptr_top;
                result = SOME_FREE;
            }
        }
        ptr = ptr -> pNext;
    }
    return (result);
}

static BOOL IsAvailableHandle (WORD handle)
{
    register struct HANDLE *ptr;

    for (ptr = pFreeEMBs -> pLowest; ptr != NULL; ptr = ptr -> pNext) {
        if (ptr -> wHandle == handle) return (TRUE);
    }
    return (FALSE);
}
```

In the next chapter all the bits and pieces get put together to make the XMSWATCH utility. In the meantime, we need to look more closely at some concrete examples of XMS extension. XMSWATCH merely uses the XMS extension facility to *snoop* on extended-memory allocations. But some programs truly *modify* XMS, for example to control XMS usage by other programs.

XMS Modification by Task Switchers for Windows Standard Mode and the DOS Shell

Readily available examples of programs that hook into XMS services to modify other programs' access to extended memory are provided by Microsoft's own task switchers for Windows standard mode and the MS-DOS Shell (in DOS 5 and higher). Windows enhanced mode also hooks XMS services, but we won't be getting into that here, except to note that the V86MMGR VxD uses the `Install_V86_Break_Point` function to install its XMS function (so that, apart from the standard JMP and three NOPs, its XMS handler consists of just one byte, the breakpoint), and that enhanced mode does not follow the XMS documentation regarding the allocation of zero-length EMBs. The remainder of this section focuses entirely on the task switchers in Windows standard mode and in the DOS shell.

Each task in these operating environments is a carefully controlled DOS session in which any number of programs, even TSRs, may be run pretty much as they would be in an ordinary DOS session without task-switching software. The division of labor differs slightly from one environment to another, but present interest centers only on the way that a task's access to XMS resources is subject to the switcher's whims. Table 14-1 shows which component is responsible for XMS handling in each of the operating environments studied here.

Table 14-1: Switcher Component Responsible for XMS Interception

Windows 3.0 standard mode	WINOA286.MOD
MS-DOS Shell	DOSSWAP.EXE
Windows 3.1 standard mode	DSWAP.EXE and WSWAP.EXE

In all cases, programs running within a task have their access to extended memory curtailed depending on parameters given in some sort of Properties list (such as the PIF settings in Windows). A session manager informs the switcher about an XMS handle (allocated from the "true" XMS server) whose corresponding EMB contains a region that the switcher is to make visible to the task. The switcher is told the address and size of that region. It is the switcher's responsibility to hook into the XMS chain and subdivide the visible extended memory according to the requirements expressed by programs running under the switcher's control.

Among the corollaries are that, if no extended memory is to be visible within the task (for example, if the Windows PIF indicates 0 KB of XMS memory), then the switcher need not install its XMS handler. Also, if the configuration specifies, for example, that only 1MB of extended memory is to be visible to a task, then even if the XMS handle passed to the switcher corresponds to an EMB that is 4MB big, the switcher only distributes the designated 1MB to programs running in that task.

The key issue here is the difference between *global* and *local* XMS handles. Global XMS handles come from the previously-installed XMS server. Local XMS handles are handed out by the task switcher. An exception is made if a client asks specifically for a zero-length EMB: XMS requires that this be passed straight through to the pre-existing XMS server, as is all subsequent use of the resulting handle, meaning that a local client may obtain a global EMB. The global *vs.* local XMS issue has some interesting complexities: Chapter 15 demonstrates that a program under DOSSWAP can allocate more extended memory through function 09h than function 08h says is free. If so, it gets a global block instead of a local block! Worse, a local client may be able resize a local block into a global one, corrupting DOSSWAP's array of handle structures in the process. Not good!

To fully understand this issue, we first need to step back in time to Windows 3.0, whose standard mode did not hook XMS in the manner set forth in the XMS documentation.

WINOA286.MOD for Windows 3.0 Standard Mode

In Windows 3.0 standard mode, WINOA286.MOD hooks XMS, but instead of hooking the existing chain of XMS handlers as prescribed in the XMS documentation, it instead hooks int 2Fh function 4310h to delude switcher clients into thinking that WINOA286 is the primary XMS server.

The XMS handles supplied to programs running under Windows 3.0 standard mode are confections of WINOA286 which bear no relationship to the XMS handles known outside Windows. Specifically, a task is allowed a maximum of 32 handles, numbered from 0001h to 0020h. Note the possible conflict: If program A1 running inside the task switcher communicates privately with program A2 loaded before the switcher, and thereby obtains an XMS handle that was originally allocated by the "true" XMS server, the global XMS handle passed from A2 to A1 may be numerically equal to one that WINOA286 has allocated to another program (B) running in the same task. When program A1 uses the global XMS handle inside the task switcher, WINOA286 will interpret the global XMS handle as one of its own, allowing possible corrupting memory to be used by program B.

To manage its handles, WINOA286 maintains two arrays. The first is just an array of 32 bytes, one for each handle, in increasing order. If the byte corresponding to a WINOA286 handle is 00h, then that handle is not allocated; if the byte is FFh, then the handle is allocated. The second is an array of structures, each representing a sub-block of the visible extended memory region and indicating whether the sub-block is currently associated with an XMS handle.

Table 14-2: Structural Representation of EMBs Presented to Clients of WINOA286.MOD

OFFSET	SIZE	DESCRIPTION	
00h	byte	00h	if block is free
		FFh	to mark end of array
		other	low 8-bits of handle if sub-block is allocated
01h	dword	physical address of sub-block	
05h	byte	lock count	
06h	word	size of sub-block in KB	

DOSSWAP, DSWAP, and WSWAP

The task switcher for Windows 3.1 standard mode has more in common with the one for the MS-DOS Shell than it has with WINOA286 for Windows 3.0 standard mode. An immediately obvious departure is that DOSSWAP, DSWAP, and WSWAP hook the XMS chain in the manner prescribed in the XMS documentation.

They also take the trouble to make sure that XMS handles given to local clients are not already allocated to global XMS users. This protection is arranged simply by following the XMS documentation. When a local client asks for an XMS handle, the switcher seeks a zero-byte allocation from the previously-installed XMS handlers. If successful, the corresponding XMS handle cannot mean anything to global XMS users and will not be allocated to them subsequently. The switcher is then free to mislead its local client into thinking that the handle is associated with a block of the requested size. Table 14-3 shows the array element used by these three switchers to relate sub-blocks of the visible extended memory region to XMS handles obtained from the "true" XMS server.

Table 14-3: Structural Representation of EMBs Presented to Clients of DOSSWAP.EXE, DSWAP.EXE, and WSWAP.EXE

OFFSET	SIZE	DESCRIPTION	
00h	word	0000h	if block is free
		FFFFh	to mark end of array
		other	XMS handle if sub-block is allocated
02h	dword	physical address of sub-block	
06h	byte	lock count	
07h	word	size of sub-block in KB	

Table 14-4 below describes exactly how the four task switchers handle each XMS function. In this extended table, a functional interface may be described in terms not just of Input and Output (sometimes broken into Success and Failure), but also by giving conditions that cause the switcher's handler simply to *Pass* the function to previously installed handlers (unchanged).

All four variations of the switcher are transparent to Get XMS Version (function 00h) and the several functions for A20 management (numbers 03h to 07h). These functions do not appear in the table.

XMS functions under WINOA286 do not preserve the flags register. This corruption is not noted explicitly in the table. XMS functions under WINOA286 and DOSSWAP are not re-entrant, contrary to the XMS specification. DSWAP and WSWAP disable interrupts for all their XMS handling, but those which call the previously installed XMS server (which may enable interrupts during its processing of the request) must also be supposed to be not re-entrant.

The task switchers generally trap XMS functions 01h, 02h, 10h, and 11h only to ensure their failure, effectively removing HMA and UMB management from the operating environment provided while task-switching is enabled. Functions 08h to 0Fh inclusive (such as the ones concerned with EMB management) are intercepted nontrivially. All other functions are passed on without change.

Table 14-4: Extension of XMS Functionality by WINOA286, DOSSWAP, DSWAP, and WSWAP

XMS function 01h - Allocate HMA

INPUT:

AH	01h

OUTPUT:

ax	0000h	
bl	91h	as if the HMA were already allocated

This listing does not apply to WSWAP, which is transparent to this function.

XMS function 02h - Free HMA

INPUT

AH	02h

OUTPUT

ax	0000h	
bl	93h	as if the HMA were not yet allocated

This listing does not apply to WSWAP, which is transparent to this function.

XMS function 08h - Query Free Extended Memory

INPUT

AH	08h

PASS (DOSSWAP ONLY):

if no free space exists in the region of visible extended memory

SUCCESS (WINOA286 AND DOSSWAP):

Table 14-4: Continued

ax	size in KB of largest free sub-block in region of visible extended memory
bl	00h (WINOA286 only)
dx	WINOA286 and DOSSWAP: total in KB of free sub-blocks in region of visible extended memory

SUCCESS (DSWAP AND WSWAP)

ax	size in KB of largest free sub-block in region of visible extended memory or largest free block reported by previously installed XMS handlers
bx	0000h
dx	total in KB of free sub-blocks in regions of visible extended memory and of extended memory reported by previously installed XMS handlers

Of the functions that are handled non-trivially by DOSSWAP, DSWAP, and WSWAP, this is the only one whose treatment varies.

In the case of DOSSWAP, the implementation of this function is oblivious to extended memory outside the EMB that contains the task's visible extended memory while even 1K of the visible extended memory remains unused by DOSSWAP's clients. If, however, a client allocates all the visible extended memory, then a repeated enquiry will be passed to the previously installed XMS handlers, which may have free memory to report, especially if extended memory was fragmented before the switching software was loaded. This will result in a global XMS handle.

Table 14-4: Continued

XMS function 09h - Allocate Extended Memory Block

INPUT

AH	09h
DX	requested size in KB - may be zero

PASS (NOT WINOA286)

if DX is zero; or

if the XMS server cannot provide a handle for an empty block; or

if no free sub-block in region of visible extended memory is sufficiently large

Table 14-4: Continued

SUCCESS

ax	0001h
bl	00h
bh	corrupt
cx	corrupt (WINOA286 only)
dx	XMS handle

FAIL (WINOA286 ONLY)

ax	0000h	
bl	A0h	if no free sub-block in region of visible extended memory is sufficiently large
	A1h	if all 32 local XMS handles defined by WINOA286 are already allocated
bh	00h	

DOSSWAP, DSWAP, and WSWAP deal with requests for empty (zero-length) blocks by passing them to the previously installed XMS handler immediately, as required by the XMS specification. Otherwise, they seek their own empty block from the XMS server, adapting this for their own purposes should enough free space exist within the extended memory region visible to the task. To previously installed XMS handlers, the handle ultimately returned in dx always describes an empty block. Only through the switcher is this handle associated with any memory.

Observe that DOSSWAP, DSWAP, and WSWAP pass the allocation request to the previously installed XMS handler if they cannot meet it from the extended memory they are making visible to the task. If extended memory was fragmented before the switcher was started (or less likely, has become fragmented since the switcher started), then the previously installed XMS handlers may be able to satisfy the request. The problem for XMS users under the control of these switchers is that they cannot, strictly speaking, determine whether their extended memory allocations are local or global.

Table 14-4: Continued

XMS function 0Ah - Free Extended Memory Block

INPUT

AH	0Ah
DX	handle

PASS (NOT WINOA286)

if the switcher has not allocated the handle DX to a client

Table 14-4: Continued

SUCCESS

ax	0001h	
bl	00h	

FAILURE

ax	0000h	
bl	A2h	WINOA286 only: if WINOA286 has not allocated the handle DX to a client
	A2h	WINOA286 only: if handle DX is locked
	ABh	not WINOA286: if handle DX is locked

For DOSSWAP, DSWAP, and WSWAP, this function involves a call to the previously installed XMS handlers in order to release the empty handle obtained for function 09h. The call is assumed to succeed.

Table 14-4: Continued

XMS function 0Bh - Move Data to or from Extended Memory Block

INPUT

AH	OBh
DS:SI	address of parameter block

OFFSET	SIZE	DESCRIPTION
00h	dword	number of bytes to transfer
04h	word	source handle
06h	dword	source offset
0Ah	word	destination handle
0Ch	dword	destination offset

OUTPUT

ax, bx from previously installed XMS handler

WINOA286 only: dx from previously installed XMS handler

The previously installed XMS handlers are always called to process this function, but some care is needed if either handle is among those obtained by the switcher for suballocation to clients. The previously installed XMS handlers know nothing of the switcher's reassignment of empty handles to sub-blocks in the region of extended memory visible to programs under the switcher's control. In these cases, the request is rephrased in terms of the "true" XMS handle to the EMB that contains the task's visible extended memory. The conversion is done in a static buffer to preserve the parameters passed at DS:SI. The implementation for WINOA286 performs this conversion correctly only if both transfer regions either start at the beginning of the visible extended memory region or are in the real-mode address space.

The switcher does not check that the source or destination region does actually lie within the sub-block associated with the source or destination handle. The implementation for WINOA286 simply assumes that the handles given in the parameter block are valid handles defined by WINOA286.

Table 14-4: Continued

XMS function 0Ch - Lock Extended Memory Block

Input

AH	0Ch
DX	handle

Pass (not WINOA286):

if the switcher has not allocated the handle DX to a client

Success

ax	0001h
dx:bx	physical address of block

Failure

ax	0000h	
bl	A2h	WINOA286 only: if WINOA286 has not allocated the handle DX to a client
	ACh	if lock count is already FFh

XMS function 0Dh - Unlock Extended Memory Block

Input:

AH	0Dh
DX	handle

Pass (not WINOA286)

if the switcher has not allocated the handle DX to a client

Success

ax	0001h

Table 14-4: Continued

FAILURE

ax	0000h	
bl	A2h	WINOA286 only: if WINOA286 has not allocated the handle DX to a client
	AAh	if block is not locked

XMS function 0Eh - Get Information about Extended Memory Block

INPUT

AH	0Eh
DX	handle

Pass (not WINOA286):

if the switcher has not allocated the handle DX to a client

SUCCESS

ax	0001h
bl	WINOA286 only: corrupt
	not WINOA286:
	number of handles still unallocated by the previously installed XMS server
bh	lock count for handle DX
dx	size in KB of sub-block associated with handle DX

FAILURE (WINOA286 ONLY)

ax	0000h

DOSSWAP, DSWAP, and WSWAP call the previously installed XMS handlers to determine how many handles remain unallocated. The call is assumed to succeed.

The value that WINOA286 returns in bl, which should be the number of XMS handles that remain free, is essentially random. The code loops through an array of data to see which handles have been allocated, but the number of iterations in the loop is the number passed to the XMS function in register CX rather than the number of entries in the array. In the case where WINOA286 fails the function (because the handle DX is not a valid handle allocated by WINOA286), no error code is loaded into register bl.

Table 14-4: Continued

XMS function 0Fh - Reallocate Extended Memory Block

INPUT

AH	0Fh
BX	new size in KB
DX	handle

PASS (NOT WINOA286)

if the switcher has not allocated the handle DX to a client

SUCCESS

ax	0001h
bl	00h

FAILURE

ax	0000h	
bl	A0h	if new block is sought but cannot be obtained
	A2h	WINOA286 only: if WINOA286 has not allocated the handle DX to a client
	ABh	if block is locked

Reduction of a block can be dealt with internally by the switcher and is always possible.

Enlargement can be processed internally if enough free space exists immediately above the block to be resized. Given this condition, the request will succeed. Otherwise, a new block of the desired size is sought so that the reallocation may be accomplished as a three-step process of allocate new, copy contents, and release old. The necessary XMS requests (functions 09h, 0Bh, and 0Ah) are issued to the whole XMS chain. The data transfer is simply assumed to succeed.

In the case of DOSSWAP and DSWAP, it may be possible that the new block is allocated by previously installed XMS handlers, so that this function converts a local XMS handle to a global one. Worse, the coding is not prepared for this possibility, with the result that the array of structures representing the sub-blocks of the task's visible extended memory becomes corrupt. This does not apply to WSWAP, which is transparent to this function.

To conclude this tour through the XMS handling in Microsoft's task switchers, observe the utter frailty of dynamic XMS extended memory usage by programs running under these task switchers. Programs that rely on XMS extended memory might expect some odd behavior when running under the DOS Shell task switcher or Windows standard mode. A program that relies on XMS or that hooks XMS might, for example, want to call int 2Fh function 4B02h to determine if it is running under a task switcher.

Table 14-4: Continued

XMS function 10h - Allocate Upper Memory Block

INPUT

AH	10h

OUTPUT

ax	0000h	
bl	B1h	as if UMBs do not exist

XMS function 11h - Free Upper Memory Block

INPUT

AH	11h

OUTPUT

ax	0000h	
bl	B1h	as if UMBs do not exist

CHAPTER 15

An Extended Memory Monitor

In this chapter many components developed in earlier chapters get put together as one seriously useful program, XMSWATCH. The utility is built from the following components:

- Chapter 4: arena.c
- Chapter 6: crtkeepc.lib
- Chapter 7: driver.asm, crtdrvr.lib
- Part 3 introduction: getxms.c, version.c
- Chapter 11: free.c, hma.c, list.c, xmsfunc.c
- Chapter 14: api.c, consult.c, handlers.c, install.c, memowner.c, monitor.c, watched.c, xms.asm, xmschain.c
- Chapter 15: main.c

The bulk of this chapter focuses on using XMSWATCH to examine different XMS environments. First, however, XMSWATCH needs a top-level module.

```
/* main.c - primary file for xmswatch.exe */

#include    <dos.h>
#include    <stdarg.h>
#include    <stdio.h>
#include    <stdlib.h>
#include    <string.h>

#include    <keepc.h>
#include    <standard.h>
#include    <sysvars.h>

/*  A description of the program - for display with the /? switch  */

static CHAR description [] = "\n\
Reports extended memory usage.\n\
\n\
  /INSTALL  retains a copy in memory to monitor XMS activity \n\
             Without this switch, the program cannot describe\n\
             extended memory that has been allocated already.\n\
\n\
The program may be run as a normal DOS executable or loaded with a DEVICE\n\
statement in CONFIG.SYS. It is recommended that it be loaded as a device\n\
driver with the /INSTALL option immediately after the main XMS server.\n";

/*  Functions and data concerned with the presence of an XMS server  */

extern void (_far *GetXMSAddress (void)) (void);
void (_far *lpXMS) (void);

/*  Installation and report generation          */
extern int Install (void);
extern void ReportVersion (void);
extern void ReportHMA (void);
extern void ReportFreeEMBs (void);
extern void ReportHandlers (void);
extern void ReportWatchedEMBs (void);

/*  A function in the style of printf to lend error messages some degree of
    uniformity */

void PutError (CHAR *, ...);

/*  Miscellany     */

extern struct SYSVARS _far * _RESIDENT lpSysvars;
static struct SYSVARS _far *GetSysvarsAddress (void);

extern WORD _far * _RESIDENT lpUMBLinkSeg;
static WORD _far *GetUMBLinkSegAddress (_segment);

extern WORD _RESIDENT wDOSVersion;

/*  ======================================================================  */

int main (int argc, char **argv)
{
    register CHAR *ptr;
    register BOOL install = FALSE;

    /*   Parse the command line. The present version admits only a /? switch
```

```
      to obtain the syntax and a /install switch (or any abbreviation) to
      load the program permanently. A likely future extension would be
      switches that allow selective reporting about the XMS environment. */

while (++ argv, -- argc) {
      ptr = *argv;
      if (*ptr == '/') {
          ptr ++;
          if (ptr [0] == '?' AND ptr [1] == '\0') {
                printf (description);
                return (0);
          }
          if (strnicmp ("install", ptr, strlen (ptr)) == 0) {
                install = TRUE;
                continue;
          }
          ptr --;
        PutError ("invalid switch - %s", ptr);
        return (-1);
      }
      else {
          PutError ("invalid parameter - %s", ptr);
          return (-1);
      }
}

/*    An XMS server such as HIMEM.SYS must be present. */

lpXMS = GetXMSAddress ();
if (lpXMS == NULL) {
      PutError ("No XMS server present");
      return (-1);
}

/*    On behalf of a routine in the program's resident section, obtain
      and record the address of DOS's internal variable table. For DOS
      versions 5 and higher, supplement this with the address of the
      variable in which the DOS kernel keeps the segment address of the
      arena header immediately at the end of the conventional memory
      chain. */

lpSysvars = GetSysvarsAddress ();
if (_osmajor >= 5)
      lpUMBLinkSeg = GetUMBLinkSegAddress ((_segment) lpSysvars);
wDOSVersion = MK_WORD (_osmajor, _osminor);
```

```
    /*   If the necessary switch has been given, attempt to install the
         program in memory to monitor future XMS allocations. */

    if (install) return (Install ());

    /*   From here on, the goal is to generate a report on extended memory
         services provided by the XMS. Deal with each XMS feature in a
         separate function. This allows the program to be extended with
         minimal disruption. */

    ReportVersion ();
    ReportHMA ();
    ReportFreeEMBs ();
    ReportHandlers ();
    ReportWatchedEMBs ();

    return (0);
}

static struct SYSVARS _far *GetSysvarsAddress (void)
{
    _asm {
              mov     ah,52h
              int     21h
              mov     ax,bx
              mov     dx,es
    }
}

static WORD _far *GetUMBLinkSegAddress (_segment spDOSData)
{
    _asm {
              xor     dx,dx
              xor     cx,cx
              mov     bx,0015h
              mov     ax,1607h
              int     2Fh
              xor     ax,ax             // prepare to return dx:ax = NULL
              cwd
              jcxz    done
              cmp     word ptr es:[bx],0005h
              jb      done
              mov     ax,word ptr es:[bx+0Ch]
              mov     dx,spDOSData
```

```
          done:
    }
}

/*  Message output  */

void PutError (CHAR *str, ...)
{
    va_list va_start (argptr, str);
    printf ("\nXMSWATCH fatal error:  ");
    vprintf (str, argptr);
    printf ("\n");
    va_end (argptr);
}
```

The full XMSWATCH program uses modules developed over the course of several chapters. Apart from assembly language stubs that provide the expected formats for device drivers and XMS handlers, all modules are in C, but some contain resident material (and therefore require special compiler switches) and some do not. A makefile is nearly essential.

```
# xmswatch.mak - makefile for xmswatch.exe

!INCLUDE <standard.mak>
!INCLUDE <keepc.mak>

TARGETS : xmswatch.exe

api.obj : api.c xmswatch.h
  $(COMPILE) $(KEEP) api.c

arena.obj : arena.c
  $(COMPILE) $(KEEP) /D _MANUAL_UMBLINK_ arena.c

consult.obj : consult.c xmswatch.h

driver.obj : driver.asm
  $(ASSEMBLE) /D "_ATTRIBUTE_=IS_CHAR_DEVICE OR IOCTL_OK" \
  /D _NAME_=XMSWWWW0 driver.asm

free.obj : free.c xmsfunc.h xmswatch.h

handlers.obj : handlers.c
```

```
hma.obj : hma.c xmsfunc.h

install.obj : install.c xmswatch.h

list.obj : list.c xmswatch.h

main.obj : main.c

memowner.obj : memowner.c
  $(COMPILE) $(KEEP) memowner.c

monitor.obj : monitor.c xmswatch.h
  $(COMPILE) $(KEEP) monitor.c

version.obj : version.c

watched.obj : watched.c xmsfunc.h xmswatch.h

xms.obj : xms.asm

xmschain.obj : xmschain.c

xmsfunc.obj : xmsfunc.c xmsfunc.h

#   Link with DRIVER.OBJ, CRTDRVR.LIB and CRTKEEPC.LIB

#   GETXMS.OBJ is also in the library directory and is assumed to be
#   current.

xmswatch.exe : api.obj arena.obj consult.obj driver.obj free.obj \
                handlers.obj hma.obj install.obj list.obj \
                main.obj memowner.obj monitor.obj version.obj \
                watched.obj xms.obj xmschain.obj xmsfunc.obj
  $(LINK) @<<xmswatch.lnk
driver+crtdrvr.lib+crtkeepc.lib+getxms+
api+arena+consult+free+
handlers+hma+install+list+
main+memowner+monitor+version+
watched+xms+xmschain+xmsfunc
xmswatch;
<<KEEP
```

A Brief Tour of Different XMS Environments

Now that the XMSWATCH program has been pieced together, readers can examine extended memory on much the same basis that MEM provides for conventional memory (or more correctly, that MEM has provided since DOS 4, this diagnostic need having been left before then to several good third-party programs). Figure 15-1 shows a typical report obtained when running only standard utilities from the DOS 5 package (not counting XMSWATCH itself).

Figure 15-1: XMSWATCH Report on Typical DOS 5 Configuration

```
XMS driver version 2.77 (Specification 2.00)

The HMA has already been allocated.

Report on Free Extended Memory:

  Free memory (KB):  3256
  Available handles:  29

  Linear Address  Size (KB)
  ---------------------------
     004D2000h      3256

Report on XMS Handler Chain:

    Entry Point     Owner
    ------------------------
     0266:00A2      HIMEM
     02B1:05E6      XMSWATCH
     C801:0CD5
     C8E6:01EB      RAMDRIVE
     C931:0FAB      SMARTDRV

Extended Memory Allocation Detected by Resident Component of XMSWATCH:

  Handles monitored:  32

  Linear Address  Size (KB)  Locks   Handle    Owner         Properties
  ---------------------------------------------------------------------
     00110000h       264       0     A71Ch    EMM386
     00152000h      1536       0     A716h    RAMDRIVE
     002D2000h      2048       0     A722h    SMARTDRV
     004D2000h      3256                                    free
```

The relevant lines in CONFIG.SYS are given in Figure 15-2. The machine has 8MB of RAM and three hard drive partitions. Readers should find it easy enough to vary the configuration to suit different hardware and different directory arrangements.

Figure 15-2: Extract from CONFIG.SYS for Typical DOS Configuration

```
comment //
device c:\dos\himem.sys
device e:\tools\xmswatch.exe /install
device c:\windows\emm386.exe noems ram=c800-efff
devicehigh c:\dos\ramdrive.sys 1536 /e          // drive F:
devicehigh c:\dos\smartdrv.sys 2048 1024
dos high umb
```

Observe that the DOS kernel is using the high-memory area, EMM386 is running the DOS session as a virtual machine, exploiting the CPU's paging mechanism to provide 160KB of upper memory, and the author seems to have no better use for his extended memory than a 2MB disk cache and 1.5MB RAM disk. Thus, three XMS clients are expected to have obtained extended memory, which should leave 29 XMS handles still available out of HIMEM's default provision of 32, and slightly more than 3MB of free extended memory on this 8MB machine.

The first line of the XMSWATCH report shows that the Get XMS Version function has been answered by an XMS server that declares itself to support version 2.0 of the overall specification but has 2.77 as its own version number. It is in fact the HIMEM.SYS version 2.77 distributed with DOS 5, but note that the XMS does not itself provide the means to identify XMS servers.

The XMS is not implemented solely by HIMEM.SYS on this machine. The section dealing with XMS handlers identifies four pieces of software that have hooked into XMS services. One of these is XMSWATCH itself, so that it may spy on all other programs' allocation of extended memory blocks from the primary XMS server.

The next XMS handler hooked into the chain belongs to EMM386. The Windows 3.1 version of EMM386, configured here with its **noems** keyword to make the maximum amount of upper memory available rather than squander some of the VM address space on an EMS page frame, uses a small amount of upper memory for its XMS handler and other material that does not need to be in low memory. This upper memory portion lies outside the DOS memory arena, hence the inability of XMSWATCH to identify the owner of the XMS handler at C801:0CD5.

The most significant modification EMM386 makes to the XMS interface implemented at first by HIMEM is to add non-trivial handling of the functions for UMB management. EMM386 lifts the CPU into protected mode so that the DOS session can be run as a virtual-8086 task with an address space that offers RAM at places where the real mode (physical) address space does not. Thus, EMM386 has the responsibility for making this upper memory available to XMS clients; it is EMM386, not HIMEM, that provides the XMS UMB functions. What the XMS clients do with the upper memory is another matter entirely. In the **dos=umb** regime, DOS itself obtains all the available upper memory through the XMS interface and then distributes this upper memory to DOS programs through the usual int 21h scheme with which these programs would obtain conventional memory.

EMM386's use of paging also carries implications for A20 handling, most notably that the physical A20 line may be left permanently enabled while a virtual A20 line can be disabled in the virtual machine simply by mapping the same 16 pages to address zero and to the 1MB boundary. In some situations, EMM386 may find itself able to present an HMA in the virtual-8086 address space where none existed in the physical address space. All in all, EMM386's interception of XMS services may be fairly substantial, involving non-trivial handling of all the functions from 00h to 07h, plus the UMB functions 10h and 11h (and also 12h, in the recent EMM386 versions that are aware of this extension defined by XMS 3.0).

XMS handlers are also installed by RAMDRIVE and SMARTDRV. Neither does anything more than help HIMEM with its curious handling (noted in Chapter 12) of XMS function 01h. Whenever HIMEM is queried about the amount of free extended memory, it records the value (in kilobytes) so that the Allocate HMA function may be failed if called with DX containing this same value and AL equal to 40h. The motivation is almost certainly to fix a bug in some piece of software that was once considered important, but since the value passed in AL is absolutely meaningless in the context of the XMS documentation, there is a possibly unwitting trade-off: Microsoft has exposed prospective HMA users to potential confusion if registers AL and DX just happen to have the problematic values when an attempt is made to allocate the HMA. RAMDRIVE and SMARTDRV treat XMS function 01h slightly differently from HIMEM in that, instead of basing their decision on a value remembered from the most recent call to XMS function 08h, they actually call the function to get what would ordinarily be the current value. Anyway, were it not for all this playfulness, RAMDRIVE and SMARTDRV would be ordinary XMS clients, with no reason to hook into the XMS handler chain.

The real value of XMSWATCH is to see exactly what use XMS clients (like RAMDRIVE and SMARTDRV) are making of extended memory. Remember, the XMS itself provides only the means to ask about the XMS resources that remain available; there is no XMS facility to enumerate those resources already in use by other programs. Had XMSWATCH not been loaded as a device driver with its **/install** switch to monitor requests for XMS allocations, it would not generally be possible to determine the handles associated with the memory blocks allocated to EMM386, RAMDRIVE, and SMARTDRV or, indeed, to identify these as the programs that have used up nearly 3MB of extended memory.

Windows 3.0 Standard Mode

As our first in-depth use of XMSWATCH, let us run it under Windows 3.0 standard mode. While obviously no longer in wide use, this environment is still worth examining.

To study Windows 3.0 standard mode, reboot using the CONFIG.SYS file from Figure 15-2, but without loading EMM386. This modification is required because the DOS extender for Windows 3.0 standard mode refuses to run if the CPU is already in protected/V86 mode. DOSX.EXE from 3.0 intends to run its child program (the Windows kernel program KRNL286.EXE) in protected mode, after arranging for many BIOS and DOS interrupts to accept protected-mode selectors instead of real-mode segments. The result of this mapping is that, provided a DOS program does not perform segment arithmetic or write into code segments, it can usually be executed in protected mode without modification but with access to the 16MB address space of the 80286.[1]

The essence of the protected mode on the 80286 and higher processors is that system code running with the highest privilege is able to restrict access to system resources in order to protect programs from one another (and indeed to protect the system from errant programs). For DOSX.EXE to get itself into this position of control, it must either start from real mode, or some orderly changeover from one privileged environment to another must be arranged through some agreed interface.

In Windows 3.0, DOSX.EXE insists that it be executed in real mode so that it can create its own protected-mode system. True, there is a widely-agreed standard, the Virtual Control Program Interface (VCPI), which allows DOS extenders to co-exist with other protected-mode software by arranging an orderly transition. Many memory managers, including the EMM386.SYS distributed with Windows 3.0, support the VCPI, but DOSX.EXE does not avail itself of the VCPI's evident benefits: Windows 3.0 standard mode is designed not to run if a memory manager (or other protected mode software) is present. (Note that in Windows 3.1, DOSX is VCPI compatible.)

Figure 15-3 shows an XMSWATCH report obtained from a DOS Prompt opened from Windows 3.0 standard mode running on a system with the configuration given in Figure 15-2, varied only by omitting the line that loads the EMM386.EXE memory manager.

Figure 15-3: DOS Session Under Windows 3.0 Standard Mode with no Memory Manager

```
XMS driver version 2.77 (Specification 2.00)

The HMA has already been allocated.

Report on Free Extended Memory:
```

---

[1] Readers should realize that Windows 3.0 standard mode is just Windows real mode running under the control of a DOS extender. Memory availability may be greater and memory management easier, but to a large extent many other aspects of Windows, especially the multitasking capabilities, are hamstrung by behavior inherited from Windows real mode.

Figure 15-3: Continued

```
Free memory (KB):  0
  Available handles:  26

Report on XMS Handler Chain:

    Entry Point     Owner
    ------------------------
     0266:00A2      HIMEM
     02B1:0712      XMSWATCH
     0364:01EB      RAMDRIVE
     03AF:0FAB      SMARTDRV

Extended Memory Allocation detected by resident component of XMSWATCH:

  Handles monitored:  32

  Linear Address  Size (KB)  Locks   Handle    Owner          Properties
  ------------------------------------------------------------------------
     00110000h      1536       0     A71Ch     RAMDRIVE
     00290000h      1024       0     A716h     SMARTDRV
     00390000h        64       1     A722h     dosx
     003A0000h         2       1     A728h     dosx
     003A0800h        16       1     A72Eh     dosx
     003A4800h      4462       1     A734h     dosx
```

Observe first that the 2MB block for the SMARTDRV.SYS disk cache has been shrunk to 1MB. This is possible because WIN.COM (and DOSX.EXE, just in case Windows is started by running DOSX directly) knows the IOCTL interface for reconfiguring that particular cache's use of extended memory. As a practical aside, note the slight advantage, if only in tidiness, that results from loading RAMDRIVE before SMARTDRV: When Windows shrinks the disk cache, the memory regained is added directly to the one large free block.

Of course, once DOSX.EXE starts, that large free block does not remain free for long. In version 3.0, DOSX allocates separate EMBs for its Global Descriptor Table (GDT), Interrupt Descriptor Table (IDT), and for its segment of code that executes in protected mode. These objects are prepared in conventional memory while the CPU is still in real mode, and are then copied to the corresponding EMBs using XMS function 0Bh.

DOSX obtains the remaining extended memory so that it can control the XMS environment seen by DOS applications launched from Windows. The visibility of extended memory to a particular DOS application depends on whether WINOA286.MOD is told about the program directly or via a Program Information File (PIF). The default, as seen above, is that DOS programs have no access to extended memory. The more general case is considered shortly, but first, there is a small matter to clean up.

The Special Case of Windows 3.0 Standard Mode and QEMM386

Devotees of Quarterdeck Office Systems' well-regarded memory manager QEMM386.SYS (generally referred to as QEMM) may wonder whether the author was in his right mind when asserting that Windows 3.0 was designed to not run in the presence of other protected-mode software. After all, one of QEMM's significant advantages over the standard EMM386 was its ability to support Windows 3.0 in standard mode. If DOSX truly insisted on the CPU being in real mode when DOSX was started, how was QEMM able to arrange otherwise?

The answer, of course, is that by the time the CPU's operating mode is tested, the DOSX.EXE program that runs under QEMM is not the same one that runs under other memory managers. In what may count as the dirtiest programming in wide circulation, QEMM *patches* the DOSX code in memory at various stages as DOSX initializes. Most notably, when QEMM detects the int 2Fh function 1605h signal that indicates standard mode initialization, it locates DOSX's use of the SMSW instruction and patches the subsequent instructions so that DOSX will not abort when it discovers that the CPU is already in protected mode. Other patches are applied when QEMM notices DOSX copying material to the three EMBs. In short, QEMM modifies the DOSX code so that it complies with VCPI (one Quarterdeck spokesperson has apparently referred to this as "we ram VCPI down Windows' throat"). Thus, the code which DOSX would ordinarily use to flip between protected mode and real mode is replaced by code that uses VCPI to arrange an orderly transition between DOSX's protected mode and QEMM's (with its virtual-8086 machine).

It is an open question whether this amounts to technical superiority over other memory managers or is just the inevitable consequence of reasonably skilled developers deciding to ignore the fetters of clean programming practice. Most things are possible if attention is directed in the right places, usually by sidestepping formal requirements. What is not in doubt is that QEMM's trickery requires an intimate knowledge of DOSX's internal organization, for example, of the locations of various items of code. In these days when industry pundits can live on consultation fees arising from fears that Microsoft introduces deliberate incompatiblities, it should be noted carefully that QEMM's ability to modify DOSX to advantage (and Quarterdeck's sales at a time when standard mode really was standard) could have been broken very, very easily, if someone had wanted to.

To see Windows 3.0 standard mode running under QEMM, reboot the machine with a CONFIG.SYS file containing QEMM386 instead of the HIMEM and EMM386 combination:

```
comment //
device e:\tools\qemm\qemm386.sys x=b000-b7ff ram
device e:\tools\xmswatch.exe /install
```

```
devicehigh c:\dos\ramdrive.sys 1536 /e   // drive F
devicehigh c:\dos\smartdrv.sys 2048 1024
dos high umb
```

The main point is that XMSWATCH is still loaded as close as possible to the primary XMS server, which in this case is QEMM386.SYS. Other points are probably best left to the apparently eternal debate about the benefits of one memory manager over another, but simply note that having the drive to produce greater memory savings in the *default* configuration means that users of QEMM with VGA display systems must *explicitly* exclude the area used for the monochrome text mode if they do not welcome the idea that their system could be brought down simply by executing **mode mono**.

Figure 15-4a: DOS Session Under Windows 3.0 Standard Mode with QEMM386

```
XMS driver version 6.02 (Specification 3.00)

The HMA has already been allocated.

Report on Free Extended Memory:

  Free memory (KB):  0
  Available handles:  54

Report on XMS Handler Chain:

    Entry Point      Owner
    ------------------------
     0266:01E5       QEMM386
     0328:0712       XMSWATCH
     C803:01EB       RAMDRIVE
     C84E:0FAB       SMARTDRV

Extended Memory Allocation detected by resident component of XMSWATCH:

  Handles monitored:  62

  Linear Address  Size (KB)  Locks   Handle     Owner         Properties
  -----------------------------------------------------------------------
     00150000h      1536       0      0002h     RAMDRIVE
     002D0000h      1024       0      0003h     SMARTDRV
     00400000h        64       1      0005h     dosx
     00410000h         2       1      0006h     dosx
     00411000h        16       1      0007h     dosx
     00415000h      3248       1      0008h     dosx
     00741000h       928       1      0009h     dosx
```

Sharp-eyed readers fluent in interpreting hexadecimal addresses will have noticed that although the map of extended memory shows no free regions, the allocated blocks do not account for all the extended memory in the physical address space (as they do in Figure 15-1).

It should be no surprise that the extended memory used by QEMM itself immediately above the HMA, that is, at address 00110000h, does not appear. In Figure 15-1, XMSWATCH detects EMM386's use of extended memory because the XMSWATCH driver is loaded between HIMEM and EMM386 and is therefore able to monitor EMM386's request for extended memory. The standard configuration provides for an XMS server (HIMEM) confined to supplying extended memory to a system that remains in real mode. More sophisticated memory management, most notably to redesign the 8086 address space by using the CPU's paging facilities in protected mode, comes from a separate program. With QEMM, the two roles are combined, which has advantages (like leaving a smaller footprint in conventional memory than is possible with HIMEM and EMM386 together) and disadvantages (like being more or less unable to return to a usable real-mode configuration).[2] The impact on XMSWATCH is that the extended memory reserved for QEMM's own use is allocated internally by QEMM and is not seen in the report.

Less obvious is that QEMM's extended memory allocations have a 4KB granularity. Thus, although DOSX has allocated just 2K for its IDT, it looks (from the difference between base addresses for the corresponding block and the next) as if QEMM has set aside 4K.

Rather more puzzling is that the jump in the sequence of handles, from 0003h to 0005h, coincides with an unaccounted 192K in the address space above the 1MB block allocated to SMARTDRV.SYS. In other words, the distance from the base of the disk cache to the start of the next known block is 192K more than the 1MB reported as the size of SMARTDRV's EMB. As might be thought from the progression of EMBs upward through the address space in chronological order of allocation, this missing memory is an artifact of QEMM's preparations to mutate DOSX into a VCPI-compliant application. For one of the few explicit indications of this dirty work, run the MEM program using the **/d** switch so that EMS handles are listed. QEMM's particular method for disbursing the same pool of memory through various interfaces (XMS, EMS, and the VCPI) means that the extended memory allocations are also seen through the EMS interface as EMS handles. The relevant part of the MEM listing is presented as Table 15-1. Note that handle 0004h was not allocated by an XMS client monitored by XMSWATCH but must either have been allocated through the EMS or by QEMM internally. Inspection of QEMM confirms the latter (and that the curious name "Win3:Stn" is Quarterdeck's confection).

---

[2] This compatibility concern must be of decreasing importance, especially to technically minded users who have the latest hardware and software. It should not be forgotten, however, that someone has to bear such things in mind for those computer users who do not have the time, desire, or means to keep up to date. Microsoft seems to be much better at this than its competitors, arguably because it is Microsoft's responsibility in exchange for the position (and substantial benefits) of contolling the standard form of the operating system. An admittedly contentious corollary is that some improvements offered by independent vendors owe a small debt to Microsoft for releasing them from having to be too concerned about backward compatibility.

Table 15-1: EMS Handles in DOS Session Under Windows 3.0 Standard Mode with QEMM386

```
Handle      EMS Name      Size
-------     --------     ------
      0                  090000
      1     HMA          010000
      2     EMB2         180000
      3     EMB3         100000
      4     Win3:Stn     004000
      5     EMB5         010000
      6     EMB6         004000
      7     EMB7         004000
      8     EMB8         32C000
      9     EMB9         0E8000
```

Non-trivial XMS Handling Under Windows 3.0 Standard Mode

To see how Windows 3.0 standard mode provides DOS applications with extended memory, it is necessary to create a suitable Program Information File (PIF). Reboot with the template CONFIG.SYS, start Windows 3.0 standard mode, run the PIF Editor, and use the file _DEFAULT.PIF as the basis for the few necessary changes. Set the Program Filename to "COMMAND.COM," the Windows Title to "DOS Prompt," and, most important, enter values for "XMS Memory." The values used for Figure 15-5 are 1024K for KB Required and 8192K for KB Limit, but the main point is that KB Required should be non-zero (which forces WINOA286 to install its XMS handler) and that KB Limit exceed the amount of physical extended memory available to DOSX. Save the edited PIF and, instead of starting a DOS session by clicking the usual icon, launch COMMAND.COM via the new PIF.

Figure 15-5: Session with Non-trivial PIF Settings Under Windows 3.0 Standard Mode

```
XMS driver version 2.77 (Specification 2.00)

The HMA has already been allocated.

Report on Free Extended Memory:

  Free memory (KB):  4433
```

Figure 15-5: Continued

```
 Available handles:  217

  Linear Address  Size (KB)
  ---------------------------
     003AB800h      4433

Report on XMS Handler Chain:

    Entry Point     Owner
  ------------------------
     1EC8:2EA0      dosx

Extended Memory Allocation detected by resident component of XMSWATCH:

  Handles monitored:  32

  Linear Address  Size (KB)  Locks   Handle    Owner        Properties
  ---------------------------------------------------------------------
     00110000h      1536       0     A71Ch    RAMDRIVE     invalidated
     00290000h      1024       0     A716h    SMARTDRV     invalidated
     00390000h        64       1     A728h    dosx         invalidated
     003A0000h         2       1     A722h    dosx         invalidated
     003A0800h        16       1     A72Eh    dosx         invalidated
     003A4800h      4462       1     A734h    dosx         sub-let
```

Note immediately that there is now just one XMS handler. It happens to be identified as belonging to DOSX but is in fact supplied by the Windows application called WINOA286.MOD, whose second segment is copied to the real-mode address space where DOSX was originally loaded.

WINOA286 does not follow the guidelines for building an XMS chain. This means that WINOA286 is unable to intercept XMS requests issued by XMS clients that were loaded before Windows. Observe that the handles defined by HIMEM and monitored by XMSWATCH mean nothing to WINOA286, which XMSWATCH has detected by noting that none of the handles known from HIMEM appear in the list of free handles built when it was mapping free extended memory. Inspection shows that WINOA286 defines 32 XMS handles numbered from 0001h to 0020h. Unfortunately for the author's credibility, XMSWATCH is a little generous in assuming the integrity of the XMS implementation and does not observe that although XMS function 0Eh indicated the existence of 217 free handles, only 32 can actually be allocated. One of the benefits of writing a book, however, is that the agent of one's embarrassment can be pilloried in public: WINOA286's implementation of XMS function 0Eh returns an essentially random value in register bl (for the count of free handles), which is not the sort of thing one anticipates, but which is apparently the sort of thing that Microsoft feels programmers should be left to discover for themselves.

Configuring SMARTDRV.SYS

As a dynamic user of extended memory, SMARTDRV may be used to probe the weaknesses of the XMS edifice. Exploration with XMSFILE lends itself very naturally to the construction of a SMARTDRV configuration program which will be handy for further probing the system. Among the tools that could be devised for testing purposes, this is one for which most of the groundwork has already been laid in an earlier chapter (when the XMSFILE driver was given the means to steal memory from the SMARTDRV.SYS disk cache). All that remains is to call the same code (in SMARTDRV.C) in a different way to cater for the different context of configuring the disk cache from the DOS command line. The resulting SDCONFIG utility resizes and reconfigures the SMARTDRV.SYS disk cache. (As noted in an earlier chapter, the newer SMARTDRV.EXE has a different interface and cannot be similarly resized except by Windows or by a program willing to pretend that it is Windows.)

```
/* sdconfig.c - primary file for sdconfig.exe */

#include    <stdarg.h>
#include    <stdio.h>
#include    <stdlib.h>
#include    <string.h>

#include    <smartdrv.h>
#include    <standard.h>

static CHAR description [] = "\n\
Resizes and otherwise configures the SMARTDrive disk cache.\n\
\n\
  SDCONFIG [newsize|MIN|MAX] [OFF|ON|RESET] [/s]\n\
\n\
Without parameters, the program reports the current cache size and status.\n\
\n\
Otherwise, specify a new cache size in KB or a new status. The /s switch\n\
directs the program to report the cache's condition after making changes.\n\
";

/*  SMARTDrive manipulations at DOS function level are handled externally -
    in SMARTDRV.C  */

extern int OpenSMARTDrive (void);
extern BOOL SupportsIOCTL (int);
extern BOOL ReadSMARTDrive (int, struct SD_READ *);
extern BOOL WriteSMARTDrive (int, struct SD_WRITE *);
extern BOOL CloseSMARTDrive (int);

void ShowStatus (struct SD_READ *);

static int sd_handle;
static struct SD_READ sd_read;
static struct SD_WRITE sd_write;

static void Cleanup (void);
static BOOL IsReallyZero (CHAR *);
static void PutError (CHAR *, ...);
static void PutNewLine (void);

int main (int argc, char **argv)
{
    register CHAR *ptr;
    register WORD newsize;
    BOOL max = FALSE, min = FALSE, sizeoption = FALSE;
    BOOL off = FALSE, on = FALSE, reset = FALSE, statusoption = FALSE;
    BOOL showstatus = FALSE;

    while (++ argv, -- argc) {
          ptr = *argv;
          if (*ptr == '/') {
              ptr ++;

              /*    A description of the program may be obtained by supplying
                    the /? switch as usual. */

              if (ptr [0] == '?' AND ptr [1] == '\0') {
                    printf (description);
                    return (0);
              }

              /*    The /s switch indicates that the cache status should be
                    shown after writing to the device to allow verification
                    of any changes. */

              if ((ptr [0] | 0x20) == 's' AND ptr [1] == '\0') {
                    showstatus = TRUE;
                    continue;
              }
```

```
        /*  No other switches are accepted. */
        ptr --;
        PutError ("invalid switch - %s", ptr);
        return (-1);
    }
    else {
        if (NOT statusoption) {
            statusoption = TRUE;
            if (stricmp ("OFF", ptr) == 0) {
                off = TRUE;
                continue;
            }
            if (stricmp ("ON", ptr) == 0) {
                on = TRUE;
                continue;
            }
            if (stricmp ("RESET", ptr) == 0) {
                reset = TRUE;
                continue;
            }
            statusoption = FALSE;
        }

        /*  A new cache size is expected as a single numerical
            parameter. */
        if (NOT sizeoption) {
            sizeoption = TRUE;

            newsize = atoi (ptr);
            if (newsize != 0 OR IsReallyZero (ptr)) {

                /*  Scale by 16 because SMARTDrive measures its size in
                    multiples of 16KB. */
                newsize >>= 4;
                continue;
            }

            /* For convenience, allow MIN and MAX keywords. */
            if (stricmp ("MIN", ptr) == 0) {
                min = TRUE;
                continue;
            }
            if (stricmp ("MAX", ptr) == 0) {
                max = TRUE;
                continue;
                }
                sizeoption = FALSE;
            }

            /*  No other parameters are accepted. */
            PutError ("invalid parameter - %s", ptr);
            return (-1);
        }
    }

    /*  Open the SMARTAAR device. */
    sd_handle = OpenSMARTDrive ();
    if (sd_handle == -1) {
        PutError ("cannot open SMARTAAR device");
        return (-1);
    }

    /*  DOS will, of course, close the now-open device when the program
        terminates, so there is no real need to close the device explicitly
        at the end of every decision path taken by the program from here on.
        On the other hand, it is undeniably beneficial to at least be aware
        that, in general, programs must allow for cleaning up activities
        once they obtain a system resource. Strictly speaking, programs
        should trap the DOS exceptions as well. */

    atexit (Cleanup);

    /*  Check that the handle actually refers to a character device
        supporting an IOCTL interface, then read the data packet that
        describes SMARTDrive's configuration. */

    if (NOT SupportsIOCTL (sd_handle)) {
        PutError ("name conflict with file or unrecognised SMARTAAR device");
        return (-1);
    }

    if (NOT ReadSMARTDrive (sd_handle, &sd_read)) {
        PutError ("cannot control SMARTAAR device");
        return (-1);
    }

    /*  In the absence of command-line options, generate a status report and
        return. */
    if (NOT (sizeoption OR statusoption)) {
        ShowStatus (&sd_read);
        return (0);
    }

    if (sizeoption) {
        /*  Now that the cache configuration is known, resolve MIN or MAX or
            check that the numerical value given for the new cache size is
            in fact within the allowed range. */

        if (min) newsize = sd_read.MinimumSize;
        else if (max) newsize = sd_read.MaximumSize;
        else {
            if (newsize < sd_read.MinimumSize
                    OR newsize > sd_read.MaximumSize) {
                PutError ("desired size %uKB is out of range (%u to %u)",
                    newsize << 4,
                    sd_read.MinimumSize << 4, sd_read.MaximumSize << 4);
                return (-1);
            }
        }

        /*  Set the parameters for the write packet. Skip the step of
            writing to the device if the cache size is not to change. */

        if (newsize != sd_read.CurrentSize) {
            if (newsize < sd_read.CurrentSize) {
                sd_write.data.size = sd_read.CurrentSize - newsize;
                sd_write.command = SD_DECREASE_CACHE;
            }
            else {
                sd_write.data.size = newsize - sd_read.CurrentSize;
                sd_write.command = SD_INCREASE_CACHE;
            }

            /* Write the new control information to the SMARTAAR device. */
            if (NOT WriteSMARTDrive (sd_handle, &sd_write)) {
                PutError ("unable to change cache size");
                return (-1);
            }
        }

        /*  If the /s switch was given, read the configuration back from
            the device for an up-to-date status report. */
        if (showstatus) {
            if (NOT ReadSMARTDrive (sd_handle, &sd_read)) showstatus = FALSE;
```

```
}
    }

    if (statusoption) {
          if (off) sd_write.command = SD_DEACTIVATE;
          else if (on) sd_write.command = SD_ACTIVATE;
          else if (reset) sd_write.command = SD_RESET;

          if (NOT WriteSMARTDrive (sd_handle, &sd_write)) {
              PutError ("unable to change cache status");
              return (-1);
          }
    }

    /*    If the /s switch was given, read the configuration back from the
          device for an up-to-date status report. */
    if (showstatus AND ReadSMARTDrive (sd_handle, &sd_read))
          ShowStatus (&sd_read);

    return (0);
}

void ShowStatus (register struct SD_READ *sdrptr)
{
    printf ("\nSMARTDrive version %u.%02u installed %s:",
              sdrptr -> MajorVersion, sdrptr -> MinorVersion,
              sdrptr -> IsActive ? "and active" : "but deactivated");
    printf ("\nCurrently using %uKB of %s memory",
              sdrptr -> CurrentSize << 4,
              sdrptr -> MemoryType == 1 ? "extended" : "expanded");
    PutNewLine ();
    printf ("\nMinimum cache size is %5uKB", sdrptr -> MinimumSize << 4);
    printf ("\nMaximum cache size is %5uKB", sdrptr -> MaximumSize << 4);
    PutNewLine ();
    printf ("\nCache has room for %u tracks", sdrptr -> TracksInCache);
    printf ("\nand presently contains %u valid tracks",
              sdrptr -> CurrentTracks);
    PutNewLine ();
    printf ("\nCache performance this session:  ");
    printf ("\n  %2u%% of sector reads filled from main cache",
              sdrptr -> HitRatio);
    printf ("\n  %2u%% of sector reads filled from intermediate buffer",
              sdrptr -> BufferRatio);
    PutNewLine ();
}
```

```
static void Cleanup (void)
{
    CloseSMARTDrive (sd_handle);
}

static BOOL IsReallyZero (register CHAR *ptr)
{
    CHAR ch;
    do {
         ch = *ptr ++;
    } while (ch == '0');
    return (ch == '\0' ? TRUE : FALSE);
}

static void PutNewLine (void)
{
    printf ("\n");
}

void PutError (CHAR *str, ...)
{
    va_list va_start (argptr, str);
    printf ("\nSMARTDRV fatal error:  ");
    vprintf (str, argptr);
    PutNewLine ();
    va_end (argptr);
}
```

Just compile this SDCONFIG.C module and the SMARTDRV.C module from Chapter 13, this time without the switches for using the DOS kernel's int 2Fh function 12h side door. Link the object modules to get the SDCONFIG.EXE program.

Fragmented Extended Memory and DOSSWAP

An immediate use of the SDCONFIG program is to test an assertion made earlier regarding the DOS Shell, by resizing the SMARTDRV.SYS disk cache in a way that fragments extended memory before running the DOS Shell. Chapter 14 provides details of how the DOSSWAP task switcher that accompanies the DOS Shell hooks into the XMS environment. It is noted there that the implementation seems to assume that all free extended memory will lie in one contiguous block when DOSSWAP is started. We can use SDCONFIG to see what happens when this assumption is incorrect.

Boot the system with a CONFIG.SYS file in which SMARTDRV.SYS is loaded before RAMDRIVE. For instance,

```
device c:\dos\himem.sys
device e:\tools\xmswatch.exe /install
device c:\windows\emm386.exe noems ram=c800-efff
devicehigh c:\dos\smartdrv.sys 2048 1024
devicehigh c:\dos\ramdrive.sys 1536 /e
dos high umb
```

Run **sdconfig min** to reduce the disk cache. This creates a hole in extended memory, as shown by the relevant section of the XMSWATCH report:

```
Extended Memory Allocation detected by resident component of XMSWATCH:

  Handles monitored:  32

  Linear Address  Size (KB)  Locks   Handle    Owner          Properties
  ----------------------------------------------------------------------
     00110000h       264       0     A71Ch     EMM386
     00152000h      1024       0     A716h     SMARTDRV
     00252000h      1024                                      free
     00352000h      1536       0     A722h     RAMDRIVE
     004D2000h      3256                                      free
```

Now start the MS-DOS Shell and check that task switching is enabled. Set the "XMS Memory" properties for the command prompt so that "KB Required" is 512K and "KB Limit" is blank. Readers who have modified this configuration to accommodate a different supply of physical memory should note that the main point is to specify that, when COMMAND.COM is started as a task under the control of DOSSWAP.EXE, the amount of extended memory visible to the task will be less than the amount recovered from the disk cache. Now start the command prompt and run XMSWATCH.

```
XMS driver version 2.77 (Specification 2.00)

The HMA has already been allocated.

Report on free extended memory:

  Free memory (KB):  512
  Available handles:  28

  Linear Address  Size (KB)
  ---------------------------
     00252000h      1024
     004D2000h       512
```

```
Report on XMS handler chain:

    Entry Point     Owner
    ------------------------
     0266:00A2      HIMEM
     02B1:0712      XMSWATCH
     C801:0CD5
     C8E6:0FAB      SMARTDRV
     CCDF:01EB      RAMDRIVE
     0A99:3082      DOSSWAP

Extended Memory Allocation detected by resident component of XMSWATCH:

  Handles monitored:  32

  Linear Address  Size (KB)  Locks   Handle    Owner         Properties
  ---------------------------------------------------------------------
     00110000h       264       0     A71Ch     EMM386
     00152000h      1024       0     A716h     SMARTDRV
     00252000h      1024                                    free
     00352000h      1536       0     A722h     RAMDRIVE
     004D2000h      3256       1     A734h     DOSSWAP      sub-let
```

Working backward, observe that DOSSWAP has obtained only the larger of the two blocks of extended memory that were free when the DOS Shell was started. The smaller block created by minimizing the disk cache remains free. DOSSWAP has hooked the XMS chain so that it can control the XMS environment seen by its child programs. In the Report on Free Extended Memory, it may be seen that when a child program (in this case, the non-resident instance of XMSWATCH) queries the amount of available extended memory, DOSSWAP answers only in terms of the one block it is sub-letting to its clients (and even then, the result is subject to the constraints imposed by the Properties settings). The answer of 512K ignores the free memory outside that block, but as the report shows, clients may allocate the extra free memory if they trouble themselves to look for it. Thus, although XMSWATCH was told of the availability of 512K, it was able to secure not just 512K from DOSSWAP but also 1MB that HIMEM knew to be free.

The general conclusion is that if a program running under the DOS Shell's task switcher allocates the amount of extended memory originally said to be free and then repeats its query about available memory, it may find that the supply has suddenly increased!

As if this were not confusing enough, it is possible to allocate in one call to function 09h more memory than function 08h declares to be free. To see this, note the address of the XMS entry point (which happens to be 0266:00A2 in the report), start DEBUG, and assemble the instructions

```
F:\>debug
-a
1968:0100 mov ah,08
1968:0102 call 0266:00a2
1968:0107 mov dx,0400
1968:010A mov ah,09
1968:010C call 0266:00a2
1968:0111 mov ah,0a
1968:0113 call 0266:00a2
1968:0118 int 3
1968:0119
-
```

Execute the first two of these instructions (a **t** command to trace through the simple MOV instruction, then a **p** command to step over the CALL). This should confirm that the XMS interface reports 0200h kilobytes of free extended memory—the 512K which DOSSWAP has been configured to present to clients. Despite the declared availability of just 0200h kilobytes, a request for an EMB of 0400h kilobytes will succeed, as shown by executing the next three instructions. To complete this stage of the demonstration, enter the **g** command at the DEBUG prompt: the remaining instructions will release the EMB.

In the case just studied, DOSSWAP finds that it cannot meet the request for 1MB of extended memory and therefore passes the XMS call to the previously installed XMS handlers. In particular, to HIMEM, which knows of the 1MB released earlier by minimizing the SMARTDRV.SYS disk cache.

A related problem occurs if the XMS client first allocates the 512KB from DOSSWAP but then tries to increase the EMB's size by using XMS function 0Fh. Restart DEBUG, but this time assemble the instructions

```
F:\>debug
-a
1968:0100 mov dx,0200
1968:0103 mov ah,09
1968:0105 call 0266:00a2
1968:010A mov bx,0400
1968:010D mov ah,0F
1968:010F call 0266:00a2
1968:0114 int 3
1968:0115
-
```

Execute the first three instructions, then check that a 512K EMB has indeed been allocated successfully. If so, proceed with the remaining instructions. DOSSWAP finds that it cannot increase the original block, so it issues an XMS request for a block of the desired size into which it may then copy the original block's contents. Unfortunately, the subsequent bookkeeping does not seem prepared for the possibility that the new block will be provided by the original XMS server. The array of structures in which DOSSWAP keeps details about the handles it provides to clients gets corrupted.

This is perhaps fortunate, because it spares clients any subtle problems that may follow from their XMS handles' sudden conversion from local to global. In the circumstances arranged here, DOSSWAP overwrites the marker that is meant to terminate the array. Since the space allowed for the array is followed by the code for XMS handling, the result of even passive inspection of the array depends on the interpretation of this code in terms of the structures DOSSWAP uses to represent XMS handles. For instance, on the machine the author has used for this experiment, running MEM shows that function 08h now declares

```
48257024 bytes available XMS memory
```

which is not only improbably large but impossible with HIMEM version 2.77, which ignores memory above 16MB. At this point, do not even try running any XMS client that may ask for extended memory allocations. Each allocation will usually cause some change to at least one element of the handle array, so if enough allocations are sought, DOSSWAP may end up overwriting its own code, now that the array is effectively unbounded. The likely result is a system crash, which adventurous readers may see for themselves by running XMSWATCH (which, remember, allocates as many zero-length EMBs as the XMS server seems able to support) instead of MEM.

Demonstration of Serious Flaws with HIMEM's Reallocate EMB

Boot the computer with a CONFIG.SYS file containing lines similar to

```
device c:\dos\himem.sys
device e:\tools\xmswatch.exe /install
device c:\windows\emm386.exe noems ram=c800-efff
devicehigh c:\dos\smartdrv.sys 2048 1024
devicehigh e:\tools\xmsfile.exe /s
devicehigh c:\dos\ramdrive.sys 1536 /e
dos high umb
```

The main points are to use HIMEM.SYS as the XMS server, to load SMARTDRV, XMSFILE, and RAMDRIVE in that order, to configure XMSFILE for global EMB allocation, and to leave at least as much free extended memory as the disk cache consumes. The resulting memory map reported by XMSWATCH should look like this:

```
Extended Memory Allocation detected by resident component of XMSWATCH:

  Handles monitored:   32
  Linear Address  Size (KB)  Locks   Handle    Owner         Properties
  ----------------------------------------------------------------------
     00110000h       264       0     A71Ch     EMM386
     00152000h      2048       0     A716h     SMARTDRV
     00352000h         0       0     A722h     XMSFILE
     00352000h      1536       0     A728h     RAMDRIVE
     004D2000h      3256                                      free
```

By default, XMSFILE allocates an empty block during its initialization. Note that HIMEM's management of EMBs assigns a base address to the empty block. The intention here is to use SDCONFIG to shrink the disk cache by 64K (or some multiple thereof, this being very important for reasons that will unfold dramatically over the next few pages). Since the cache in the example starts with size 2048K, the desired new size is 64K less than that, 1984K. Running **sdconfig 1984** gives a report similar to this:

```
SMARTDrive version 3.13 installed and active:
Currently using 1984KB of extended memory

Minimum cache size is  1024KB
Maximum cache size is  2048KB

Cache has room for 233 tracks
and presently contains 73 valid tracks

Cache performance this session:
  52% of sector reads filled from main cache
  11% of sector reads filled from intermediate buffer
```

Run XMSWATCH to confirm that reducing the cache creates a small hole directly underneath the empty block allocated to XMSFILE.

```
Extended Memory Allocation detected by resident component of XMSWATCH:

  Handles monitored:  32

  Linear Address  Size (KB)  Locks   Handle    Owner         Properties
  ----------------------------------------------------------------------
     00110000h       264       0     A71Ch     EMM386
     00152000h      1984       0     A716h     SMARTDRV
     00342000h        64                                      free
     00352000h         0       0     A722h     XMSFILE
     00352000h      1536       0     A728h     RAMDRIVE
     004D2000h      3256                                      free
```

When a small text file is copied to the XMSFILEX clipboard, the Reallocate EMB function (as implemented by HIMEM.SYS) will find first that it cannot enlarge the XMSFILE EMB in place, but second that it can expand downward. This takes up the slack created when the disk cache was shrunk.

```
Extended Memory Allocation detected by resident component of XMSWATCH:

  Handles monitored:  32

  Linear Address  Size (KB)  Locks   Handle    Owner          Properties
  --------------------------------------------------------------------
     00110000h       264       0     A71Ch     EMM386
     00152000h      1984       0     A716h     SMARTDRV
     00342000h        64       0     A722h     XMSFILE
     00352000h      1536       0     A728h     RAMDRIVE
     004D2000h      3256                                     free
```

Now if SDCONFIG is used to restore the disk cache to its maximum size, SMARTDRV's own code will ask the XMS server to resize the memory block to 2048K. When HIMEM gets to work, it will find that SMARDTRV's block cannot simply be grown in place (XMSFILE's block is in the way). Instead, HIMEM will have to allocate a new block (entirely disjoint from the existing SMARTDRV EMB) and then copy the contents of the disk cache to this new block.

```
Extended Memory Allocation detected by resident component of XMSWATCH:

  Handles monitored:  32

  Linear Address  Size (KB)  Locks   Handle    Owner          Properties
  --------------------------------------------------------------------
     00110000h       264       0     A71Ch     EMM386
     00152000h      1984                                     free
     00342000h        64       0     A722h     XMSFILE
     00352000h      1536       0     A728h     RAMDRIVE
     004D2000h      2048       0     A716h     SMARTDRV
     006D2000h      1208                                     free
```

These reports from XMSWATCH should help readers fix in mind what is happening as the various extended memory blocks are reallocated. The event to concentrate on is the relocation of the disk cache when it is maximised after XMSFILE has been used to block a more straightforward expansion.

Readers are cautioned not to use the HIMEM.SYS version 3.07 distributed with Windows 3.1 for this demonstration, because when Reallocate EMB actually has to allocate a fresh block and transfer data the 80386-specific implementation of the function in this version may crash the computer. (On the other hand, ignoring this caution makes for an even more dramatic demonstration of serious flaws in the Reallocate EMB function!)

Now repeat the experiment but set an XMSFILE cluster size of 16KB and reduce the SMARTDRV disk cache by 48K, not 64K. After copying text to the clipboard, the memory map should look like this:

```
Extended Memory Allocation detected by resident component of XMSWATCH:

  Handles monitored:  32

  Linear Address  Size (KB)  Locks   Handle    Owner        Properties
  ----------------------------------------------------------------------
     00110000h       264       0     A71Ch     EMM386
     00152000h      2000       0     A716h     SMARTDRV
     00346000h        16       0     A722h     XMSFILE
     0034A000h        32                                    free
     00352000h      1536       0     A728h     RAMDRIVE
     004D2000h      3256                                    free
```

Before attempting to restore the disk cache to its maximum size, copy COMMAND.COM, SDCONFIG.EXE, and XMSWATCH.EXE to the RAM drive, start a new command processor, and

```
F:\>set comspec=f:\command.com
```

It should not be necessary, but readers may care to run CHKDSK on a hard drive to make certain that the disk cache contains data. With F: as the default drive, running **sdconfig max** causes the disk cache to be relocated as before, as may be confirmed by running XMSWATCH from the RAM drive. Unfortunately, the Reallocate EMB coding error observed in Chapter 12 means that the contents of the disk cache are not copied, as will be all too clear if the SMARTDRV code in low memory believes the cache contains directory and FAT information for drive C:

```
F:\>dir c:

 Volume in drive C has no label
 Directory of C:\

File not found
```

Do not even consider writing to any cached disk drive at this point, but by all means, take a few moments to imagine the reaction of someone who did not know that this situation had been prepared in a controlled environment. To recover (from the apparent lack of a hard disk, but not necessarily from the implications), run **sdconfig reset** to empty the extended memory cache, press Ctrl-C at the DOS command line to flush the DOS kernel's **buffers** system, then finally exit from the command processor to restore the COMSPEC environment variable.

Adventurous readers may care to repeat the experiment without availing themselves of the pre-planned escape hatch. The most likely consequence is that when COMMAND.COM next tries to reload its transient portion, it will be unable to open the COMMAND.COM file on the cached drive. This is why the escape hatch takes the trouble to use a copy of the command processor and to set COMSPEC appropriately. Without the escape provision, the resident portion of COMMAND.COM will halt the system when it finds that continuation is impossible. Readers with the DOS 6.20 upgrade may have noticed documentation that describes this sudden loss of the command processor as a symptom of suspect memory hardware that fails suddenly (having passed the BIOS's apparently insufficiently rigorous tests at startup). The author has no wish to commit the logic failure of declaring that the main or real cause of the problem is the reproducible cause isolated here. However, the DOS 6.20 exercise seems infused with the notion that reported (controversial) problems with one component of DOS 6.0 were due to hardware failure and were not the natural consequences of defective software, as so many had supposed. At least in the case of HIMEM and memory hardware, a little scepticism does not seem out of place.

If the cache size is reduced by 16KB instead of 48KB, then instead of having none of the cache get copied to the new EMB, all but the last 16K will be transferred; if the cache is not full, there may be no untoward consequences. Otherwise, with just one track or so of corrupt cache data, how long will it be before an observer encounters the inevitable calamity and what is the likelihood that the cause will then be identified correctly?

Let it be absolutely clear that this is no ordinary programming error. Because the XMS does not provide software with the means to discover the distribution of allocated extended memory blocks, an XMS client cannot know in advance whether a proposed resizing will involve a reallocate-and-copy. This means that *all* calls to Reallocate EMB must be regarded as potentially dangerous. At least for HIMEM.SYS versions 2.xx, software may side-step the problem by allocating and resizing extended memory blocks in multiples of 64KB only, but for the newer versions of HIMEM which implement 80386-specific code for XMS 3.0, there is no work-around, which is why all the demonstrations in this book use the HIMEM.SYS distributed with DOS 5.

Compare this situation with the bug described in Chapter 10. In that case, HIMEM may compute a negative size for the XMS pool, not having realized that by delaying its int 15h hook, it gives software the chance to invalidate earlier calculations about how much memory HIMEM will eventually seize for later distribution among XMS clients. There is something particular about the circumstances of this bug, with the result that the problem is avoidable, and, when it is eventually corrected, it will not leave any lasting implications.

This is true of many, if not most, problems with software. The effects are limited to the performance of the software concerned. In very rare cases, there may be consequential damage: for instance, if a bug in a disk utility causes the software to corrupt data structures that are important to the integrity of the file system. In general, however, the user can obtain a fix from the vendor, preferably at no cost, which resolves the problem for all practical purposes.

By contrast, a defect in an extant implementation of an interface reaches further. It affects developers of third-party products, who are in no position to guarantee that their end-users run the latest version of the implementation on which their products depend. Short of the publication of this text, it remains possible that developers who pay considerable sums to Microsoft for supposedly up-to-date information on DOS and Windows programming may find themselves facing a great deal of difficult work tracing a bug reported by a customer who happens, for one reason or another, not to be using the very latest version of HIMEM.

The coding of XMS function 0Fh in HIMEM.SYS means that a portion of an interface, whose adoption by programmers has been encouraged and which is often described as the product of a committee of industry leaders, cannot be used safely. Everyone accepts the inevitability of coding errors in complex software and accepts that the methods available for reasonable cost of checking software cannot catch all problems, but again, this is an *interface* problem that Microsoft has been aware of and has silently fixed but has seemingly not made developers aware of. Silently fixing the newer versions of HIMEM does not solve the problem, because programs are likely to encounter the newer, fixed, versions on their machines and in their own test labs, but the older, broken, versions are more likely to be running on customers' machines. The apparent failure to alert developers to this problem is a disgrace and makes irresistable the suggestion that the drive to increase software sales has exceeded the capacity of people, organizations, and at least one multi-billion dollar corporation to behave responsibly.

Windows Enhanced Mode

Running XMSWATCH inside a virtual machine constructed by Windows in its Enhanced mode shows how HIMEM (configured to offer just eight handles to ensure the report's brevity) is supplanted by another XMS implementation:

```
XMS driver version 2.05 (Specification 2.00)

The HMA has already been allocated.

Report on Free Extended Memory:

  Free memory (KB):  1024
  Available handles:  3

  Linear Address  Size (KB)
  ---------------------------
     ????????       1024

Report on XMS Handler Chain:

    Entry Point     Owner
    ------------------------
     0266:00A2      HIMEM
     02B1:05E6      XMSWATCH
     C801:0CD5
     C8E6:01EB      RAMDRIVE
     C931:0FAB      SMARTDRV
     07AE:0045      win386

Extended Memory Allocation detected by resident component of XMSWATCH:

  Handles monitored:  8

  Linear Address  Size (KB)  Locks   Handle    Owner        Properties
  --------------------------------------------------------------------
     00000000h         0        0     A73Ah    win386      consumed
     00000000h         0        0     A740h    win386      consumed
     00110000h       264        0     A71Ch    EMM386
     00152000h      1536        1     A716h    RAMDRIVE
     002D2000h      1024        1     A722h    SMARTDRV
     003D2000h       533        0     A728h    win386
     00457400h      3747        0     A72Eh    win386
     00800000h         0        0     A734h    win386      consumed
```

Note first that the version number has changed. The V86MMGR VxD inside WIN386 replaces HIMEM as the XMS server and reports itself, perhaps confusingly, as version 2.05. It is essential to good communication between the original DOS session and the virtual machine that this replacement be effected by threading a new handler onto the existing XMS chain, not by installing an int 2Fh routine to change the XMS calling point. This means that WIN386 has first option on all XMS calls, even those issued by programs loaded *before* it.

Note also that Windows obtains from the preexisting server all available extended memory and also all the handles. Some of this extended memory is used for running the Windows kernel. Most of it is reserved for Windows applications and 1MB has been left for allocation by programs running in virtual machines. Observe that (at least in its default configuration) Windows does not allow programs running in the DOS box to lock their

XMS handles and discover the linear address of the memory concerned. This gives Windows complete freedom to page this memory differently should the need arise, or even to implement a virtual memory system in which the XMS is simulated by disk storage. Windows, of course, is a sophisticated operating environment that takes PC programming into another world, but it starts from DOS and runs DOS programs. Increasingly, the DOS world is just a virtual machine.

PART 4

Disk Management

Sooner or later, almost every PC user considers making an exact copy of a floppy disk. If only the files and directory structure need to be copied, then these can be copied to a temporary directory on a hard disk or RAM disk, from where they may be dispatched to floppy disks. Usually the XCOPY utility is used, especially if the files are organized in directories.

The thought of copying sector by sector is often rejected for practical reasons even though it should be faster at duplicating a disk that is nearly full. The problem is that DISKCOPY—the standard DOS utility for this purpose—is constrained to use only conventional memory. As a result the source disk must be reread for each duplicate and the operation inevitably requires several more disk changes than seem necessary. In circumstances that do not offer the choice of copying just the file structure—say, because a binary duplicate is needed for safety before attempting data recovery or disk repair—DISKCOPY's blindness to all that free extended memory can be very frustrating.

A disk duplication utility is a natural candidate for a serious demonstration of extended memory use by a DOS program. Its main requirement is temporary storage space—lots more than DOS usually offers its applications. Data is simply read from disk in pieces of some suitable size and copied as a block to whatever storage is available. Later the process is reversed (using a different disk, presumably). The program should be small and fast. To this end, the XMS facility for block moves to and from extended memory is ideal. The disk duplication program gets its extra memory without having to buy a complete protected-mode environment.

These arguments have been rehearsed before. So too have the methods for storing data in and retrieving it from extended memory. The real difficulties, as ever, lie elsewhere. . .

CHAPTER 16

Low-level Disk Support

First, a disk-duplication facility (whether it uses extended memory or any other form of intermediate storage space) needs to bypass the DOS file system and read/write sectors off the disk.

Absolute Disk Read and Write

From the very beginning, DOS has provided a way to undercut the file system and work with disks at sector level. The utility programs CHKDSK, DISKCOMP, and DISKCOPY all require this facility for their normal function and DEBUG offers a command for reading and writing "absolute sectors," ostensibly to help overcome a calamity that cannot be repaired mechanically by CHKDSK.

In essence, the absolute disk services are front-ends to the procedures the DOS kernel uses ordinarily when calling block device drivers to read or write sectors as part of a bigger operation at file level. The kernel validates the drive, locates the device driver responsible for access to that drive and then calls the driver's Strategy and Interrupt routines. The advantage to applications of using int 25h or 26h rather than calling the device driver directly is that the kernel also performs the bookkeeping tasks typical of an int 21h function. An internal DOS stack is used, the same one used for most int 21h functions above 0Ch. The InDOS flag is incremented to offer some protection from interruption. In a network environment or when Windows is running, the kernel also takes the trouble to issue the necessary critical section signals—but these are fine points.

Table 16-1: Interrupts 25h (Absolute Disk Read) and 26h (Write)

INPUT (OLD STYLE):

AL	0-based drive number
CX	number of sectors
DX	starting sector
DS:BX	address of transfer buffer

Table 16-1: Continued

INPUT (NEW STYLE):

AL	0-based drive number
CX	FFFFh
DS:BX	address of parameter block

OFFSET	SIZE	DESCRIPTION
00h	dword	starting sector
04h	word	number of sectors
06h	dword	address of transfer buffer

SUCCESS:

cf	set

FAILURE:

cf	clear
al	device driver error
ah	error code in style of int 13h, typically reconstructed from device driver error:

device driver error description and code		int 13h style error code
write-protect violation	00h	03h
not ready	02h	80h
CRC error	04h	10h
seek error	06h	40h
sector not found	08h	04h
general failure	0Ch	02h
anything else		00h

Under DOS 4 and higher, some error combinations are simulated because the input parameters are rejected before calling the responsible device driver:

ax 0201h if invalid drive

Table 16-1: Continued

0207h	if old style of input is used for drive with more than 64K sectors

CORRUPTS:

flags	in particular, interrupts are always enabled
stack	the flags placed on the stack by the CPU as part of its normal response to the **int** instruction are left there, and need to be removed by the caller

In DOS versions below 4, no attempt is made to preserve registers.

The new style of input (which allows for 32-bit sector addressing) is available in DOS versions 4 and higher and also in *some* OEM implementations of earlier versions (most notably Compaq 3.31). DOS continues to support the old style, but note that from DOS 4 onward, this old style cannot be used if the drive's capacity exceeds 64K sectors—even if all sectors involved in the request are within range, that is, even if the operation does not involve 32-bit logical sector numbers. An exception is made for drives with the special value of FFh sectors per cluster (as understood through the DPB, typically because a floppy drive has not held a disk since the default block device driver in IO.SYS was initialized).

In DOS 5 the absolute sector operations are integrated properly with the BUFFERS system (the various sector caches with which DOS aims to reduce its need for disk access during file-level operations). Before a write request is passed to the device driver, DOS searches the BUFFERS for sectors that are about to be invalidated and discards them. When a request to read sectors returns successfully from the device driver, DOS checks to see whether any of these sectors just read from disk ought really to have been taken from the BUFFERS system instead. These are of course the same steps DOS takes when reading or writing a stretch of whole sectors as part of a file—a sector accessed recently enough to be still in memory is more current than the copy on disk and would indeed have been on disk were it not for DOS's buffering. DOS 5 is the first version to take proper stock of this point in connection with interrupts 25h and 26h.

DOS 4 introduced a scheme (retained by DOS 5 and 6) in which requests for a single sector near to the last such request could be filled from a look-ahead buffer. The number of sectors buffered this way is set through CONFIG.SYS as the second parameter for the **buffers** directive. This cache is operated by the kernel at the level of its device driver calls. The current contents of the look-ahead buffers are invalidated before proceeding with an absolute disk read or write.

In all DOS versions there exist error conditions in which AX is not loaded with a sensible error value. The conditions, which all concern the drive number passed in AL, are

- AL = FFh;
- AL describes an invalid drive (versions below 4);
- AL describes a remote drive (but with an exception described below for DOS 4);
- AL describes a local drive that is available to other computers via the int 2Ah interface.

The consequences vary. For versions below 4, ax is unchanged from the value passed as the interrupt's input. For the more recent versions, ax is the error code returned by the last unsuccessful call that returned a proper error code. In the earlier DOS versions, the first three of these four circumstances also corrupt one word in memory. Extended error information is loaded in response to some of these error conditions but not generally. It is probably wise to regard extended error information as corrupted by a call to int 25h or int 26h.

DOS 4 allows absolute disk operations on remote drives as long as the installable file system (IFS) driver has bit 11 clear in its attribute word to indicate that the storage device is local. Skipping ahead a little, this means that the native FAT file system has been replaced but the DOS kernel still associates the drive with a DPB and block device driver. This is for DOS 4 *only*; it was dropped (along with all of IFSFUNC) from subsequent versions.

The entry code that switches to an internal DOS stack and saves the caller's registers does not clear the global variable that distinguishes normal DOS processing from calls that are being handled on behalf of a remote machine. A result is that if the most recent DOS call before the int 25h or int 26h was redirected through the server facility (int 21h function 5D00h), then virtual drives are regarded as invalid.

As noted in Table 16-1, int 25h and 26h both leave a word on the stack—the flags pushed there by the processor during the **int** instruction—which must be popped off by the caller.

Clearly this interface is not without its problems, despite recent revision. Even putting those aside, the absolute sector services are not powerful enough to be of real use in real situations. When a low-level facility is used for disk access, it is probably because the usual machinery of the device driver interface has failed in the first place, at least for the disk in question. In such situations the int 25h and int 26h front end to the device driver Read and Write commands simply does not offer enough control. They are too closely tied to offering file system support; in situations where information about the disk format is lost or can no longer be presumed accurate (typically because the disk is being formatted), this level of disk support is not low enough.

At the same time, the BIOS int 13h level usually isn't approrpiate either. It is not that access at int 13h level is intrinsically undesirable, but that the programming exercises where it is truly the best level must be few and far between, and very specialized. Operating system support is generally preferable, not least because it may allow the same code to work with storage media not supported by int 13h.

To a large extent, the DOS Generic IOCTL interface to block devices (int 21h function 440Dh) removes the need for access at int 13h level. To fully understand the often-overlooked generic IOCTL block-device services, we will first need to take a lengthy detour to examine drives and disks.

Hard Disk Partitioning

A standard scheme has evolved for dividing hard disk storage into distinct partitions to allow the one piece of (possibly expensive) hardware to be used for different and perhaps incompatible purposes. An extreme case is a computer set up with two operating systems, neither of which has the means to tap into the other's file system. In practice hard disk partitions are rarely so independent of one another. For instance, DOS offers a redirector interface (typically used for network support) through which it may acquire at least a limited ability to work with an alien system's files. In this case a unified file structure can be presented to the user, but partitioning leaves each system free to handle questions of file control in its own way in its separate area of the same storage device.

Still less independent are partitions that belong to the same operating system. Historically, DOS users have been obliged to partition hard disks larger than 32MB irrespective of their desire to share their hardware with another operating system. The problem arose because logical sector numbers were manipulated as 16-bit quantities inside the DOS kernel, through a few application interfaces and in many undocumented internal structures that applications had started using freely. It was presumably expedient to lend DOS 3.30 the ability to recognize more than one DOS partition per hard disk rather than embark on a fundamental kernel reorganization with all its implications at higher levels.

Ultimately, built-in support for disks larger than 32MB without resorting to partitions came with DOS 4. Even developers who choose not to support older versions must still deal with the consequences of hard disk partitioning, for though it may no longer seem necessary, many users regard partitioning as an important convenience. DOS 5 not only continues to support multiple DOS partitions, but has also brought improvements—increasing flexibility and laying a stable foundation for future file system enhancements.

The consequence of allowing multiple DOS partitions is that each physical hard disk may give rise to many separate DOS drives. All such drives seem physical to DOS, in the sense that they have independent file systems. Each has its own boot sector, File Allocation Tables (FATs) and root directory that precede the area available for storing files, and a change on one drive (such as creating a file) has no consequences for the other. They are as independent as drives that do indeed belong to different physical devices.

In practice then, drives C and D may belong to different physical hard disks or may simply be different partitions of the same disk. No interface with the DOS kernel allows such questions to be settled (strictly speaking, it is not even possible to establish at this level that a drive corresponds to a hard disk rather than a RAM disk or some other storage device).

The Partition Table

Information about disk partitioning is held, not surprisingly, in the very first sector. When IO.SYS is initialized it expects to find a partition sector at the start of each hard disk that the BIOS reports as fitted to the machine. In each of these a partition table containing four entries is located toward the end. Each entry is 10h bytes long. Since room must be left for the signature word AA55h at the end of the 512-byte sector, the first entry consulted is at offset 01BEh (200h - (4 * 10) + 2) in the sector. In principle, a partition table entry has the structure shown in Table 16-2. Note that the start of the partition seems to be specified in two ways: absolutely, in terms of the int 13h parameters needed to read the partition's first sector and relatively, in terms of a sector count from the beginning of the disk.

Table 16-2: General Structure of Partition Table Entry

OFFSET	SIZE	DESCRIPTION
00h	byte	distinguishes the "active" or "bootable" partition
01h	byte	head (register dh for int 13h) on which partition starts
02h	word	cylinder/sector combination (register cx for int 13h) on which partition starts
04h	byte	identifies operating system or file system, or otherwise indicates reason for partition's existence (see below)
05h	byte	head (register dh for int 13h) on which partition ends
06h	word	cylinder/sector combination (register cx for int 13h) on which partition ends
08h	dword	start of partition, counted in sectors from beginning of disk
0Ch	dword	size of partition, measured in sectors

The start of the partition seems to be specified in two ways: absolutely, in terms of the int 13h parameters needed to read the partition's first sector, and relatively, in terms of a sector count from the beginning of the disk.

In addition to the partition table the first sector of the hard disk may also contain boot code. On most systems, the BIOS loads this sector into memory for execution only if it has not been able to boot from a floppy disk. Likewise it is ordinarily only the first hard disk that is bootable. The main task of the boot code is of course to locate the operating system somewhere on disk and begin loading it. The standard arrangement is that one and only one partition on the first hard disk be marked "active" or "bootable" by setting the value 80h in the first field of its partition table entry. For other partitions the field should be zero. Note, however, that this is a property of the replaceable boot code and is of no concern to DOS. A likely generalization is a partition of size zero whose table entry serves merely to direct the bootstrap sequence to another hard disk. The first field is well-placed to go in register dl as the int 13h drive number.

The system ID at offset 04h is the one by which different operating systems recognize partitions as their own, that is, as having the expected internal structure. Table 16-3 shows the meanings assigned to various system IDs by the partitioning utility FDISK supplied with DOS; as explained in the next few pages, the different "PRI DOS" values support 12-bit *vs.* 16-bit FATs, and volumes > 32MB.

Table 16-3: Partition IDs Known to FDISK

ID	TYPE
01h	PRI DOS
02h	XENIX
03h	XENIX
04h	PRI DOS
05h	EXT DOS
06h	PRI DOS
07h	HPFS
64h	NOVELL
75h	PC/IX
DBh	CP/M
FFh	Table

FDISK simply reports other values (of which there are many) as "Non-DOS." The table demonstrates well enough that a variety of systems follow the same convention for partition table entries. It should perhaps be observed that the standard partition scheme has various supersets in which the partition table contains 8 or 16 entries, again running up to the signature word at the end of the sector.

Primary DOS Partitions

An operating system may claim more than one system ID. In DOS's case the values 01h, 04h, and 06h denote "primary" partitions of various kinds, in which DOS (or at least its default block device driver) expects to find the low-level structures involved in supporting the DOS file system—a boot sector, one or more FATs, and a root directory to be followed by the data area itself. The different system IDs have specific implications for these structures:

ID	SIZE OF FAT ENTRY	SIZE OF LOGICAL SECTOR NUMBER	PARTITION CAPACITY
01h	12-bit	16-bit	32MB
04h	16-bit	16-bit	32MB
06h	16-bit	32-bit	2GB

Two parameters affect the partition capacity. In all variants of the DOS file system, sectors are allocated in multiples commonly known as "clusters." These are tracked in the File Allocation Table (FAT). The size of the FAT entry determines the number of clusters that can be allocated for data storage in the file system. With 12-bit entries, there can be only a little fewer than 4K clusters. With 16-bit FAT entries, there can be nearly 64K clusters. DOS requires that the number of sectors per cluster be a power of 2 that can be described in one byte. A quick multiplication

64K clusters * 128 sectors per cluster * 512 bytes per sector = 4GB

gives a naive capacity much greater than indicated above for the partitions with IDs of 01h and 04h. This is where the second parameter governing DOS partitions comes into play, for when these first two partition types were defined, certain DOS structures within the partition were constrained to use 16-bit sector addressing. As a consequence of this, no matter how what the cluster size, these partitions are designed to hold at most 64K sectors. At the expected ½K per sector, this gives a maximum capacity of 32MB. With the newer partition type (introduced for DOS 4), which has 06h for its ID, there is no practical constraint on the width of variable used to index sectors, and the full naive capacity should apply. It will be shown later that IO.SYS understands this to be the maximum capacity, but in fact, the calculation breaks at a higher level: DOS 5 kernel code assumes no more than 64 sectors per cluster, thereby producing the published maximum partition size of 2GB.

The partitioning scheme allows a DOS primary partition to lie anywhere on the physical hard disk. This is shown schematically in Figure 16-1. Observe that the disk starts with the sector containing the master partition table and that the primary partition starts with a DOS boot sector.

When IO.SYS initializes, it looks at the available hard drives, trying to discern how many DOS partitions are present and where they reside on the disk. As shown in Table 16-4, only some of the fields from the corresponding entry in the master partition table (shown earlier) are relevant.

Figure 16-1: Primary DOS Partition Within Hard Disk

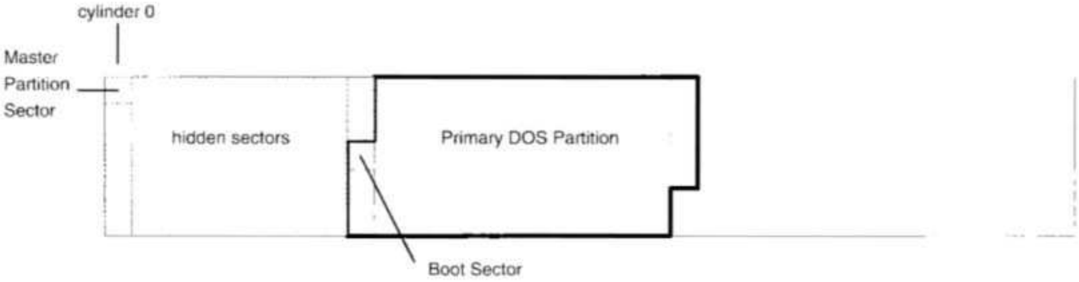


Table 16-4: Partition Table Fields Used by IO.SYS to Locate Primary DOS Partition

OFFSET	SIZE	DESCRIPTION
00h	byte	Active partition indicator: In version 5, primary DOS partitions with a value of 80h or higher in this field are assigned drive numbers before others on the same disk. For more information on this, refer to the section headed "Drive Numbering." DOS versions lower than 5 recognize only one primary DOS partition per hard disk, so this field is irrelevant to them.
04h	byte	Operating system flags: IO.SYS does not differentiate the three values that indicate primary DOS partitions. Indeed, any information they might be thought to contain about the size of FAT entries and so on is completely disregarded.
08h	dword	Start of partition: This locates the partition, counted in sectors from the beginning of the disk. At higher levels, this field is adopted as the number of "hidden sectors."
0Ch	dword	Size of partition, measured in sectors. Partitions smaller than 64 sectors are ignored.

Note especially that IO.SYS does not differentiate the three values that indicate primary DOS partitions. Any information they might be thought to contain about the size of FAT entries and so on is completely disregarded. Of course, IO.SYS is interested to know that the system ID is one of the expected values for a primary DOS partition, but it doesn't care which one in particular it is.

Of course, the system ID or other fields in the partition table entry for a primary DOS partition may be consulted at other levels, both lower and higher. For instance, if the partition is marked "active," then the partition sector's boot code is likely to use the int 13h parameters at offsets 00h to 03h as the easiest way to load the DOS partition's own boot sector. IO.SYS also needs to examine the partition's boot sector—for every primary partition, not just the bootable one—but it calculates the int 13h parameters from the number of hidden sectors. Incidentally, if the boot sector resides in a cylinder numbered higher than 1024 (a limit imposed by the int 13h interface), then the partition is ignored.

The key point here is that the file system need refer to a position on disk only by counting its distance in sectors from the start of what it regards as the physical disk. The DOS kernel (MSDOS.SYS) is entirely unaware of the so-called hidden sectors. This allows it to think of the disk—indeed of any block storage device—as an array of sectors with no structures other than those it imposes to support a file system.

The hidden sectors become relevant at a lower (device driver) level where the file system's logical sector numbers have to be decomposed into cylinder, head, and sector for the int 13h disk services. If logical sector number 0 is the first physical sector of the disk, as happens with floppy disks, then the reduction is a straightforward series of divisions. Remembering that physical sectors are numbered from 1 but that heads and cylinders number from 0, these calculations are easily written in C as

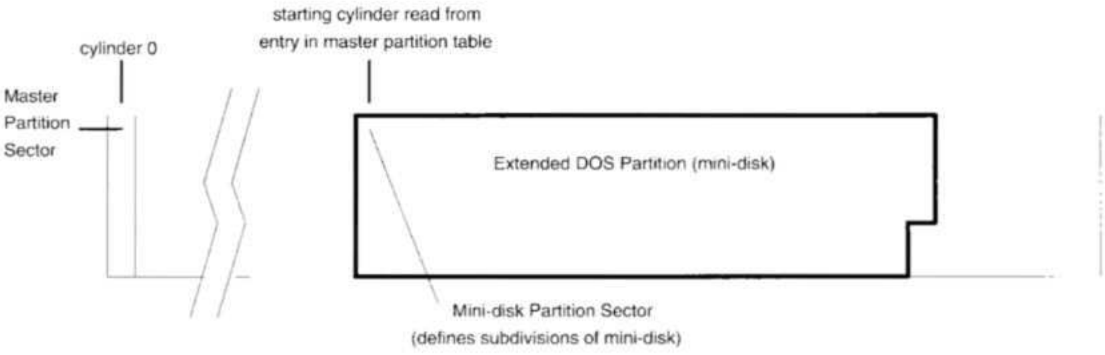
```
sector = logical_sector % sectors_per_track + 1
head = (logical_sector / sectors_per_track) % heads
cylinder = (logical_sector / sectors_per_track) / heads
```

where it helps the symmetry (as such) if `heads` are thought of as `tracks_per_cylinder`. A primary partition displaced from the beginning of the physical disk is accommodated by the simple expedient of adding the hidden sectors to the logical sector number immediately before the default block device driver starts the divisions. The catch, for early DOS versions anyway, is that restrictions on sector addressing apply not just to the partition, but to the sum of the partition's capacity and the number of hidden sectors. As long as DOS could support only 16-bit sector numbers, this constrained primary DOS partitions to lie wholly within the first 32MB of a physical disk.

Extended DOS Partitions

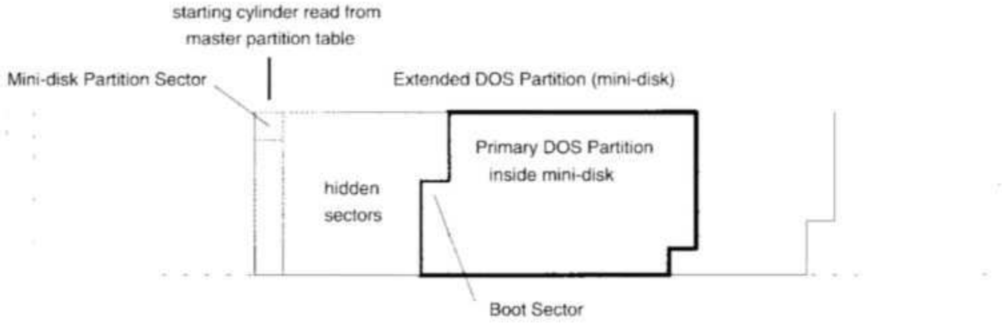
The proper solution to this problem is to remove the constraint of dealing with logical sector numbers as 16-bit quantities. As noted earlier, however, this has implications for software at higher levels, so pending a full reworking for DOS 4 (which caused no small trouble to the vendors of many disk utilities), an intermediate solution had to be found. Figure 16-2 shows how this was done by imagining a disk within a disk. An extended DOS partition is not itself associated with a DOS drive. Instead of beginning with a boot sector, like a primary DOS partition, the extended partition begins with another partition sector.

Figure 16-2: Extended DOS Partition Within Hard Disk


starting cylinder read from
entry in master partition table
cylinder 0
Master
Partition
Sector
Extended DOS Partition (mini-disk)
Mini-disk Partition Sector
(defines subdivisions of mini-disk)

In effect the extended partition is treated as a "mini-disk" that may be further subdivided along the same principles that apply to the whole disk. Thus Figure 16-3 shows a primary DOS partition inside the mini-disk from the sketch in Figure 16-2.

Figure 16-3: Primary DOS Partition Within Mini-disk


starting cylinder read from
master partition table
Mini-disk Partition Sector
Extended DOS Partition (mini-disk)
Primary DOS Partition
inside mini-disk
hidden
sectors
Boot Sector

Note that the hidden sectors extend from the start of the mini-disk, not the physical disk. The same reduction to cylinder, head, and sector for int 13h access can be made as before, then at the last moment, all that is needed is a correction for the starting cylinder of the mini-disk. In DOS 3.30 this has the powerful effect of allowing the reduction to be performed on 16-bit sectors. The partition itself plus the hidden sectors might be limited to 32MB, but the cylinder correction allows that 32MB to lie almost anywhere on the physical disk.

The whole idea is recursive. In addition to containing a primary partition of its own, the mini-disk may also contain an extended partition—another mini-disk which may support even futher subdivision. Ultimately, the recursion takes an extended DOS partition in the master partition sector and extrudes a chain of fake primary DOS partitions, each of which is adopted as a DOS drive.

Each time IO.SYS processes an entry denoting an extended partition, its only interests are to verify that the partition is large enough to hold something non-trivial and to locate the next partition table, that is, to continue the recursion. The extended partition table is therefore interpreted with slightly different emphasis, as shown in Table 16-5.

Table 16-5: Partition Table Fields Used by IO.SYS to Locate Extended DOS Partition

OFFSET	SIZE	DESCRIPTION
01h	byte	This is the head used to read the partition sector at the start of this extended partition.
02h	word	This is the cylinder/sector combination for the partition sector at the start of this extended partition. The cylinder component is extracted and saved as the starting cylinder of the mini-disk.
04h	byte	05h
0Ch	dword	This gives the partition's size, measured in sectors. An extended partition smaller than 64 sectors ends its chain, on the basis that it is too small to contain a primary partition.

Drive Numbering

Under DOS 5 hard disk partitions are inspected for adoption as drives in the following order:

1. *One primary DOS partition from the master partition table of each hard disk fitted to the machine.*

   Hard disks are processed in ascending order of int 13h drive number (and in steps 2 and 3 also). Each partition table is examined upward in two passes, since priority is given to partitions with a value 80h or higher in the first field of their table entry (that is, to active partitions).
2. *One extended DOS partition from each hard disk fitted to the machine—to extract all the primary partitions within.*

   The chain of extended partitions is followed recursively. At each branch point the mini-disk partition table is searched upward in two passes, first for a primary DOS partition to adopt as a drive and second for an extended partition to support further recursion. Recursion stops if a mini-disk partition table contains no entry for the next level, if an entry exists but describes a partition smaller than 64 sectors, or if the next mini-disk partition sector cannot be read. Curiously, IO.SYS does not require mini-disk partition sectors to end with the signature word AA55h expected of the master partition sector.

3. *All remaining primary DOS partitions on each hard disk fitted to the machine.*

   Each partition table is examined upwards in two passes, again to give priority to active partitions. The search moves on to the next hard disk only after all primary DOS partitions in the master partition table have been set up as drives.

The third step is entirely new to DOS 5. The first step has been generalized substantially. Earlier DOS versions recognized one primary partition on each of the first two hard disks only. DOS could use additional hard disks only through extended partitions.

Floppy drives are assigned numbers (letters at user level) before hard drives are inspected. IO.SYS is able to adopt only as many floppy drives as the BIOS reports, up to a maximum of four. The first two drive letters are always assigned to floppy drives. If necessary, any shortfall is made good by constructing logical floppy drives. It is well-known that a machine fitted with only one floppy drive has both A and B available for application-level access to the lone physical device. DOS versions earlier than 4 assume the presence of at least one floppy and respond to a diskless system by constructing logical drives A and B as before, but whose shared physical device is nonexistent. DOS 5 offers improved recognition of this case where no floppy drives are installed (it checks both the BIOS equipment word returned by int 11h and the number of installed floppy drives returned by int 13h function 08h), but the current implementation of logical drives to deal with a genuine absence of floppy drives exposes IO.SYS to unchecked errors capable of crashing the system in some admittedly contrived situations. This whole area—not just the preceding alarm—is taken up in detail later under "Logical Drive Mapping."

Starting with DOS 5, drive C is guaranteed to be the lowest DOS drive corresponding to a hard disk partition, a decision possibly motivated by the incredible number of installation programs that simply aren't prepared for any other arrangement. If more than two floppy drives have been assigned, IO.SYS renumbers its internal drive tables, shifting the excess floppy drives to the end of the list. Thus, a machine that had floppy drives A, B, and C with two hard disks D and E under DOS 4 would find itself with E: as a floppy drive if booted with DOS 5. Although IO.SYS ensures that the boot drive number ultimately available via int 21h function 3305h (Get Startup Drive) is updated if necessary, no attempt is made to reorder the array of BPB pointers (BIOS Parameter Block; one for each drive unit) returned when MSDOS.SYS issues the Initialize Device command (device driver function 0). For a little while, therefore, the DOS kernel has a very strange idea of which drive is which, but fortunately the information dereferenced from these pointers is not regarded as immutable—a BPB rebuild is requested before any drive access.

Boot Sectors

The partitioning scheme carves out an area of the hard disk that can be treated on a logical basis similar to a whole floppy disk—so much so, that for all practical purposes, the term "hard disk" is often used without mentioning that it is only a logical subdivision of the physical hard disk. Only at the lowest level need it be remembered that for hard disks, the reduction from logical sectors to physical disk parameters also requires knowledge of hidden sectors and possibly a correction for the mini-disk starting cylinder. By leaving such details behind, the system is free to think of a disk as beginning with a boot sector that may hold more information about the disk's format. The boot sector's general form, both for hard disks and floppies, is shown in Table 16-6.

Table 16-6: Structure of DOS Boot Sector

<table>
<tr><th>OFFSET</th><th>SIZE</th><th>DESCRIPTION</th></tr>
<tr><td>00h</td><td>03h</td><td>an intra-segment JMP instruction</td></tr>
<tr><td>03h</td><td>08h</td><td>the OEM name and version number</td></tr>
<tr><td>0Bh</td><td>19h</td><td>a BIOS Parameter Block (BPB)
<table>
<tr><th>OFFSET</th><th>SIZE</th><th>DESCRIPTION</th></tr>
<tr><td>00h</td><td>word</td><td>bytes per sector</td></tr>
<tr><td>02h</td><td>byte</td><td>sectors per cluster</td></tr>
<tr><td>03h</td><td>word</td><td>reserved sectors</td></tr>
<tr><td>05h</td><td>byte</td><td>number of FAT copies</td></tr>
<tr><td>06h</td><td>word</td><td>maximum number of root directory entries</td></tr>
<tr><td>08h</td><td>word</td><td>total sectors, or 0 to denote use of wider field at offset 15h</td></tr>
<tr><td>0Ah</td><td>byte</td><td>media descriptor</td></tr>
<tr><td>0Bh</td><td>word</td><td>sectors per FAT</td></tr>
<tr><td>0Dh</td><td>word</td><td>sectors per track</td></tr>
<tr><td>0Fh</td><td>word</td><td>heads</td></tr>
<tr><td>11h</td><td>dword</td><td>hidden sectors</td></tr>
<tr><td>15h</td><td>dword</td><td>total sectors, if field at offset 08h is zero</td></tr>
</table></td></tr>
<tr><td>24h</td><td>byte</td><td>int 13h number (register dl) for boot drive</td></tr>
<tr><td>25h</td><td>byte</td><td>used by standard boot code to hold head number (register dh) when calling int 13h</td></tr>
<tr><td>26h</td><td>byte</td><td>the signature value 29h indicates validity of the media identifiers in the remaining fields</td></tr>
<tr><td>27h</td><td>dword</td><td>serial number</td></tr>
<tr><td>2Bh</td><td>0Bh</td><td>volume label</td></tr>
<tr><td>36h</td><td>08h</td><td>file system description</td></tr>
</table>

It should be understood that the boot sector layout given here is a device driver structure, is not strictly meaningful to the DOS kernel, and as such is not a property to be expected of all DOS drives. In principle (though it has probably never been implemented in practice), a DOS drive can be supported on non-standard media without a boot sector in anything like this form. What is required is that the device driver be able to return a BPB structure at initialization and when one is requested after a negative Media Check. For the moment however, consider the default block device driver and standard storage supported via int 13h.

Hard Disk Boot Sectors

Starting with DOS 5, IO.SYS subjects hard disk boot sectors to some preliminary plausibility checks:

- The jump instruction at offset 00h must be either a JMP NEAR (opcode E9h) or JMP SHORT (opcode EBh), the latter requiring a NOP to make up the three bytes of the field;
- The media descriptor at offset 0Ah in the BPB on disk must be ≥ F0h;
- The number of bytes per sector given at offset 00h in the BPB on disk must be 0200h;
- The number of sectors per cluster given at offset 02h in the BPB on disk must be a power of 2.

If any of these conditions are not satisfied, then IO.SYS configures the partition as a device unit (so it will ultimately become a DOS drive), but the representation built for the disk is a default, based wholly on information already obtained from the partition table or from querying the BIOS about the disk drive's physical structure. In addition, IO.SYS disables access to the drive, awaiting the deliberate action of software that establishes the disk's true format. This plausibility check is probably not implemented as a defensive measure but instead offers the opportunity to replace the native DOS file systems based on 12-bit and 16-bit FAT entries while still retaining the default block device driver in IO.SYS for low-level disk operations.

A primary reason to construct a device driver unit but disable access is that the partition might not have been formatted yet; for example, the FORMAT utility calls the generic IOCTL service to enable the access flag (int 21h function 440Dh minor code 47h, described later) after it has successfully initialized the structures required to support a DOS file system on the partition.

OEM Name and Version

DOS 5 is also the first version that does not tie boot sector recognition to the OEM name at offset 03h. This is an important freedom because if a disk's boot sector has an unknown

OEM name, the other fields describing the disk's characteristics are disregarded. Instead the disk is assigned a default format based essentially on its total capacity (as before, but typically not with the provision for disabling access). In versions 3.30 for instance, IO.SYS accepted the disk format only if the OEM field started with the letters 'IBM'. DOS 4 extended the list to 'IBM', 'MSDOS', and 'OS2'. The objective is presumably to avoid boot sectors formats with known problems, and to this end DOS 3.30 and DOS 4 also checked the last characters of the OEM field for a version number.

It is clearly unacceptable that third-party format programs be rendered ineffective for not having the "right" name—certainly not if DOS is meant to work straight "out of the box." DOS 5 resolves the issue by concentrating solely on the version number assumed to conclude the OEM field. Specifically, the last character should be a single-digit minor version number, preceded by a period and further back by digits denoting the major version. With this interpretation

- major versions that are multiples of 10 other than 10 itself or 20 are regarded as unknown and access to the drive is disabled (in the sense described earlier);
- with the exception of version 2.0, all versions numbered below 3.1 are regarded as unknown.

Accepted Hard Disk Formats

If the boot sector meets the criteria for recognition, then fields in the BPB are accepted as accurate descriptions of the disk layout—up to a point. The BPB maintained by IO.SYS in memory for a disk does not necessarily match the BPB on disk. The importance of understanding this cannot be overstated: for all disk access, it is the BPB in memory that counts.

Two FAT copies are assumed, regardless of the value given in the BPB on disk. Less important, the media descriptor for hard disks is taken as F8h, again independent of the BPB on disk. The number of sectors per track and the number of heads are taken as drive characteristics returned by int 13h function 08h, not from the BPB on disk. Likewise, the number of hidden sectors is adopted from the relevant partition table—as it should be, since these sectors lie outside the storage space used by the DOS drive. In contrast, the total capacity in sectors is read from the BPB in preference to the value in the partition table, unless the BPB specifies zero (in both the word and dword fields).

That IO.SYS does not adopt a particular BPB field does not mean that the value on disk is meaningless, just that IO.SYS takes no notice of it. For instance, the standard boot code must calculate the physical parameters from which to load at least the first few sectors of IO.SYS and therefore needs to know the number of hidden sectors. It reads the field in the BPB—after all, the code must be small and is hardly in a position to hunt for fields in a partition sector.

The signature byte at offset 26h may have the magic value 29h—to denote an extended BPB that runs on to provide a serial number, a volume label, and a file system ID. More generally, the magic value indicates a modern format (the standard boot code, for instance, does not consult the 32-bit field for total sectors unless it also finds the magic number). More detail is given in the "Media Identifiers" section, but for now it should be observed that IO.SYS adopts the identifiers from the boot sector if it sees the magic value. In practice, the media identifiers kept in memory by IO.SYS are not consulted for hard disks.

IO.SYS calculates the number of whole sectors available for data, on the basis that the storage space is devoted to some number of reserved sectors, followed by the two FATs and a root directory. If the equivalent number of clusters is less than 0FF6h, then the DOS file system will be able to cope with 12-bit FAT entries; if not, then 16-bit FAT entries are required. The drive is invalidated (beyond resuscitation via the Access Flag) if the number of clusters deduced for a recognized disk format overflows 16 bits.

For the sake of completeness, observe that both DOS 4 and DOS 5 make a special case for modern disk formats (that is, ones with the magic value 29h) that indicate *no* FAT copies. In this case, IO.SYS adopts the BPB from disk without alteration, save only that DOS 5 retains the sector capacity from the partition table if the value in the BPB is zero. Calculations related to FATs are redundant to this special case and are skipped.

Unrecognized Hard Disk Formats

An ignominious fate awaits any hard disk whose boot sector DOS dismisses as having an unknown format. The disk characteristics adopted for the remainder of the session depend entirely on the properties already obtained from the partition table and from int 13h function 08h (Get Drive Parameters). The BPB in the boot sector is completely distrusted. Of course, this behavior is an important defence if the partition has not yet been formatted as a DOS drive, in other words, if a DOS boot sector with a valid BPB really hasn't been written into place.

As noted already for recognized hard disk formats, some fields in the BPB on disk are ignored anyway, but this treatment of unknown formats is especially inflexible. The following disk characteristics are imposed uniformly:

Bytes per sector	512
Reserved sectors	1
Number of FAT copies	2
Root directory entries	512
Media Descriptor	F8h

The number of sectors per cluster and the size of FAT entry are determined from the total capacity in sectors according to the following rules:

TOTAL CAPACITY IN SECTORS	EQUIVALENT IN BYTES	SECTORS PER CLUSTER	SIZE OF FAT ENTRY
≤ 0FF5h * 8	≤ approx 16MB	8	12-bit
≤ 64K * 4	≤ 128MB	4	16-bit
≤ 64K * 8	≤ 256MB	8	16-bit
≤ 64K * 16	≤ 512MB	16	16-bit
≤ 64K * 32	≤ 1GB	32	16-bit
≤ 64K * 64	≤ 2GB	64	16-bit
≤ 64K * 128	≤ 4GB	128	16-bit

The number of sectors per FAT is deduced by solving an equation. The problem is that the FAT must have enough sectors to hold as many entries as there are clusters in the disk's data area, whose size is in turn affected by the number of sectors in each FAT copy. Let

b = the number of reserved sectors at the start of the disk (boot sectors)

f = the number of sectors per FAT

r = the number of sectors allowed for the root directory

x = the number of sectors available for storing file data

t = total disk capacity in sector

then the disk's logical structure gives the relationship

$$t = b + 2f + r + x$$

A FAT consisting of f sectors can represent at most $256f - 2$ clusters of data (using 16-bit FAT entries and allowing for the first two FAT entries which do not correspond to clusters in the data area), so the FAT contains no wasted sectors if and only if

$$\left[256(f-1)-2\right]s < x \le (256f-2)s$$

where s denotes the number of sectors per cluster. The solution, given for the benefit of those who write software that formats disks or inspects boot sectors (as part of a disk repair service, for instance), is

f = the least integer greater than or equal to $\dfrac{t-b-r+2s}{256s+2}$

which is very nearly the same algorithm implemented by Microsoft. With 12-bit FATs, the solution is complicated by the way that FAT entries need not fit neatly into sectors and is not taken up here, although it is similar.

Floppy Disk Drives

With fixed hard disks there is by definition no distinction to make between the disk and its drive—at least, it is possible to imagine a "drive" with the same properties as the disk (ultimately some routine working at a low level has to face up to all this logical contrivance). For floppy disk drives, the storage medium is not fixed, and, indeed, different disks inserted into the same drive at different times may have very different formats. However, the drive itself has physical characteristics that are constant for the session.

As a normal part of its initialization, IO.SYS categorizes the available floppy disk drives. For this purpose, it relies on the BIOS to report the drive's capabilities. Int 13h function 08h (Get Drive Parameters) returns the maximum supported values for the cylinder, head and sector parameters. In the context of assigning drive types, it is sometimes helpful to know whether or not the BIOS supports "change-line" detection (i.e., the drive can detect whether the drive door is open) for the drive in question. One indirect inference is available since the low-capacity 5¼" drives are unable to provide a change-line facility.

The drive classification is seen at user level as the "form factor" known to DRIVER.SYS and the CONFIG.SYS **drivparm** statement. Table 16-7 shows the form factors that IO.SYS uses for floppy drives and the defining characteristics (the number of cylinders and the maximum number of sectors allowed on each track). For each case, the table also indicates the usual disk capacity for the drive.

Table 16-7: Relevant Factors for Determining Floppy Drive Form Factor

FORM FACTOR	CYLINDERS	SECTORS PER TRACK	TYPICAL DISK CAPACITY	COMMENTS
0	40	≤ 9	360K	also adopted if CMOS failure defeats the int 13h function 08h query or if the (old) BIOS offers neither a query function nor the ability to detect disk changes
1	80	15	1.2MB	also adopted if the BIOS does not offer a query function but can detect disk changes
2	80	9	720K	
7	-	-	1.44MB	any situation not covered by other form factors
9	80	36	2.88MB	

Form factor 7 serves as the "other" category, although in practice this means that the drive offers 80 cylinders at 18 sectors per track, giving a maximum capacity of 1.44MB. For each form factor, IO.SYS prepares a default disk format (see Table 16-8)—the standard arrangement of file system structures for a disk with the maximum capacity supported by the drive.

Table 16-8: Default Disk Characteristics for Floppy Drive Form Factor

FORM FACTOR	CAPACITY	SECTORS PER CLUSTER	ROOT DIRECTORY ENTRIES	MEDIA DESCRIPTOR	SECTORS PER FAT	SECTORS PER TRACK	HEADS
0	360K	2	112	FDh	2	9	2
1	1.2MB	1	224	F9h	7	15	2
2	720K	2	112	F9h	3	9	2
9	2.88MB	2	240	F0h	9	36	2

Form factor 7 is not listed in the table: because this form factor is the repository for unknown drives, a default disk layout has to be calculated from the characteristics returned by int 13h function 08h. The maximum capacity in sectors is just the product of the number of cylinders, the number of heads and the maximum number of sectors allowed on each track. The sectors per cluster are chosen to be the least power of 2 that permits 12-bit FAT entries for a disk with the maximum capacity just calculated. If one sector per cluster suffices, then 224 entries are allowed in the root directory; otherwise, space is set aside for 240.

Floppy Disk Boot Sectors

In the normal course of reading from and writing to floppy disks, these default formats play no role, provided that a reliable format description can be obtained from the floppy disk's boot sector. To be accepted, a floppy disk boot sector must pass the following tests, similar to those applied for hard disks:

- The jump instruction at offset 00h must be either a JMP NEAR (opcode E9h) or JMP SHORT (opcode EBh), the latter requiring a NOP to make up the three bytes of the field; to support older formats, the first byte may be 69h.
- The media descriptor at offset 0Ah in the BPB on disk must be ≥ Fh.

Just as with hard disks, however, some elements of the disk's own description of its format are countermanded. For instance, the number of FATs is always taken to be two, whatever the value in the corresponding BPB field. Readers who doubt the truth of this assertion (and have perhaps been doubting it ever since it was advanced for hard disks) are invited to test it with a disk editor. Take a newly-formatted 360K disk, make the few alterations required for a third FAT (even to the point of changing the media descriptor bytes to F0h for unknown format), exit to DOS, remove the disk from its drive, and execute the **dir** command for the drive in question. The resulting critical error forces the DOS kernel to request a BPB rebuild on the next access, so restore the disk and retry—only to find the third FAT being treated as the root directory.

The complete picture is that IO.SYS enforces the following characteristics for floppy disks, ignoring the values given in the boot sector's BPB:

Bytes per sector	512
Reserved sectors	1
Number of FAT copies	2
Hidden sectors	0

In addition, the number of root directory entries *is* read from the BPB on disk, but only modulo 256, so fiddling with the floppy disk boot sector in the hope of increasing the root directory's capacity beyond 240 would be a loser from the outset.

Reliance on the Media Descriptor

Suppose that the first sector is read but does not pass the minimal test for recognition as a boot sector. The most likely reason these days is surely corruption, be it by accident or by deliberate action of a computer virus. Without an acceptable boot sector at the start of the disk, DOS looks to the first byte of the second sector, expecting to find a "media descriptor." This, in combination with the known drive characteristics, may suffice to identify the disk as having one of the default formats. If not, the exercise is abandoned. This usually results in a critical error message attributing the problem to an "Invalid media type" or a "Non-DOS disk."

Originally the media descriptor served to distinguish the few formats understood by DOS's first version. Standards adopted later do not fit into the scheme easily. The use of F9h to describe both the 720K and 1.2MB formats is unfortunate and clearly presents recognition problems. Worse than this, the most recent formats (1.44MB and 2.88MB) have no place in the table at all. Corrupting the boot sector renders such disks unreadable at any level that depends on the device driver recognizing the disk format.

This last point returns the analysis to the original problem: the device driver Read and Write commands (and, by extension, any low-level interface that relies on them, as do interrupts 25h and 26h) are too closely tied to offering file system support. In situations where information about the disk format is lost or can no longer be presumed accurate (typically because the disk is being formatted), this level of disk support is not low enough.

Table 16-9: Deduction of Characteristics for Floppy Disk with Unrecognized Boot Sector

	CAPACITY	SECTORS PER CLUSTER	ROOT DIRECTORY ENTRIES	SECTORS PER FAT	SECTORS PER TRACK	HEADS
media descriptor < F8h	invalid media type					
media descriptor = F9h, form factor = 2	720K	2	112	3	9	2
media descriptor = F9h, form factor = 7 or 9, drive can detect disk change	720K	2	112	3	9	2
media descriptor = F9h, form factor ≠ 2, 7 or 9, drive can detect disk change	1.2MB	1	224	7	15	2
media descriptor ≠ F9h, form factor = 2	invalid media type					
media descriptor = F8h or FCh	180K	1	64	2	9	1
media descriptor = F9h or FDh	360K	2	112	2	9	2
media descriptor = FAh or FEh	160K	1	64	1	8	1
media descriptor = FBh or FFh	320K	2	112	1	8	2

The Default Block Device Driver

Programmers who attempt serious disk work should understand that DOS and its users (mostly) see disk drives not in terms of the hardware units themselves but in terms of the logical units constructed by device drivers. Previous sections have given some idea of how the default block device driver configures the units through which DOS accesses the *standard* disk storage.

Even for those few exercises where it is really essential to access the storage by direct communication with the hardware or at least through the BIOS's int 13h services, it is quite likely that the programmer will need to know the correspondence between the BIOS view of the drives and the DOS view.

The IO.SYS Drive Tables

It may help to present the structure in which IO.SYS keeps all its knowledge about the block device units.

IO.SYS maintains a table for each logical unit of the default block device driver. These tables are kept in a linked list, and int 2Fh function 0803h (see Table 16-11) returns a far pointer to the head of the list; use the link pointer at offset 0 to traverse the list. DRIVER.SYS uses int 2Fh function 0801h (see Table 16-11) to add drive tables for new units onto the list.

Table 16-10: Table Maintained for Each Logical Unit of the Default Block Device Driver

OFFSET	SIZE	DESCRIPTION	
00h	dword	far pointer to the next table, or FFFFh as low word if this table is the last	
04h	byte	drive number required in register dl by int 13h	
05h	byte	0-based DOS drive number for this unit	
06h	19h bytes	BIOS Parameter Block (BPB) representing current contents of drive	
1Fh	byte	file system flags, ostensibly for hard disk partitions	
		00h	indicates use of 12-bit FAT entries
		40h	indicates use of 16-bit FAT entries
		80h	denotes a disk or partition that cannot be supported because of some physical property, typically size, but which has nonetheless been adopted as a unit of the default block device driver—attempting access to such a drive always returns the error Not Ready
20h	word	number of times the Open Device command has been received for this drive without a matching Close Device	
22h	byte	device type—also known as form factor:	
		00h	floppy disk drive, 40 cylinders, ≤ 9 sectors per track
		01h	floppy disk drive, 80 cylinders, 15 sectors per track
		02h	floppy disk drive, 80 cylinders, 9 sectors per track
		05h	hard disk

Table 16-10: Continued

		07h	any floppy disk drive not covered by the other form factors
		09h	floppy disk drive, 80 cylinders, 36 sectors per track
23h	word	status flags for the drive:	
		0001h	non-removable media
		0002h	change-line detection supported by BIOS
		0004h	disk BPB (at offset 06h) locked
		0008h	good track layout (sectors are equally sized)
		0010h	shares whole physical device with other logical units
		0020h	active unit for shared physical device
		0040h	disk change detected
		0080h	set DASD before format
		0100h	disk reformat detected
		0200h	read/write disabled
25h	word	number of cylinders supported by the drive—for hard disk partitions, this is the number of cylinders spanning the partition	
27h	1Fh bytes	BPB representing the default disk format for the drive	
46h	byte	low 8 bits of cylinder number for last successful int 13h call to read, write, or verify	
47h	dword	for drive with removable media: system clock count after last successful disk access for drive with non-removable media: mini-disk cylinder correction to complete reduction of drive parameters from logical to physical	
4Bh	0Ch bytes	11-byte volume label with a null terminator in the 12th position	
57h	dword	serial number	
5Bh	09h bytes	8-byte file system label with a null terminator in the 9th position—the default block device driver offers "FAT12" and "FAT16", each padded with spaces to make up eight characters.	

The structure given here applies only to DOS 4 and higher. In earlier versions, the structure is smaller (there being no need for the fields concerned with media identification) and elements are arranged differently.

The IO.SYS Unit Tables are not necessarily linked in ascending order of DOS drive number. DOS 5 reserves C for the first hard disk, but in the initial assignment of drive numbers, a system whose BIOS reports three or four floppy drives at startup will have D or E as the first hard disk. In such cases the tables are renumbered (refer to the field at offset 05h) but the links between tables are not altered.

The 0-based DOS drive number at offset 04h also serves as the unit number. There are two indications that this identity is intended (or at least well-established). First, the implementations of the Logical Device Map functions of the device driver interface assume the identity. Second, the identity is preserved by the DRIVER.SYS utility when adding new units. Some disk compression products disrupt this relationship when swapping DOS drive letters for their users' convenience.

One exceptional case is created by IO.SYS itself. In DOS versions 4 and 5, if the BIOS reports no floppy drives at startup, drives A and B will both be unit number 00h and there will be no unit number 01h. This has some potential danger in the contrived event that IO.SYS receives a request to operate on unit number 01h, since many routines in IO.SYS simply assume the existence of a Unit Table for the unit number involved in a request.

An imbalance of Open Device and Close Device calls for a given unit (see offset 20h) indicates that the corresponding drive's contents are in use at file system level and should not be allowed to change without raising an alert. Note that the default block device driver heeds the difference only if at least one floppy disk drive is known to have change-line capability. It is not prepared to rely on the open/close count without also having the necessary ROM BIOS support to ensure notification of disk changes from the hardware. Moreover, the DOS kernel issues the Open/Close Device commands to block device drivers only if SHARE is installed.

After each successful int 13h call to read, write, or verify, the low eight bits of the cylinder number (register ch) are remembered at offset 46h. If the heads have not moved to another cylinder since the last access, then the 15ms head settle time which IO.SYS normally arranges for floppy disk writes is reduced to 1ms when reading or verifying.

As a sort of virtual change-line support, IO.SYS records the system clock count (at offset 47h) after each successful disk access. In the absence of any other indications, a disk is assumed to be unchanged if the same drive was used successfully no more than two seconds earlier (two seconds apparently being the shortest possible time for a human to change disks).

For drives with nonremovable media, this timing ruse is not required and the opportunity is taken to use the field at offset 47h for another purpose. It is treated as two words. In DOS 5 the first should have the value 0001h, essentially for historical reasons. The second is a cylinder correction, arising typically as the starting cylinder of the mini-disk in which this unit is a primary partition. Refer back to "Hard Disk Partitioning" for more detail. DOS 5 recognizes that the whole physical disk is a special case of mini-disk starting at cylinder zero, but in earlier versions the first word has the value FFFFh for a true primary partition and 0001h for a mini-disk partition.

Int 2Fh Access to the IO.SYS Unit Tables

As happens so often with objects deep within the system, the value of knowing about them is greatly enhanced by the existence of an interface through which they may be accessed by programs. Equally often, it seems that such interfaces are not documented. This is not unreasonable in itself, especially in the present case, where the Unit Tables have changed with passing DOS versions.

In fact, a facility to locate the Unit Tables appeared only with DOS 4. It is implemented by the simple expedient of appending a function to a pre-existing int 2Fh function 08h API (described in Table 16-11) by which IO.SYS allows DRIVER.SYS to configure extra units for the default block device driver (or, from a user's point of view, add new floppy drives or create multiple logical drives corresponding to a single physical drive).

Table 16-11: IO.SYS Int 2Fh API to Support DRIVER.SYS

External Driver Function 00h—Availability Check

INPUT:

AX	0800h

OUTPUT:

al	FFh

External Driver Function 01h—Add External Driver

INPUT:

AX	0801h
DS:DI	address of Unit Table for new unit

CORRUPTS:

ax, bx, si

The Unit Table provided via this function is added to the end of the existing chain. It is the caller's responsibility to set the DOS drive number at offset 05h in the Unit Table before adding the new unit. If any existing units use the same drive at int 13h level, then they become shared units and the new unit is marked as both inactive and shared. Change-line support is inherited from shared units but not transferred to them.

Table 16-11: Continued

External Driver Function 02h—Pass Call from External Driver

INPUT:

AX	0802h
ES:BX	address of request header

CORRUPTS:

flags

stack	the flags placed on the stack by the CPU as part of its normal response to the **int** instruction are left there

This function is in fact reached for all values of AL in the range 02h ≤ AL < F8h, with the exception in DOS 4 and higher of function 03h.

When the DRIVER.SYS interrupt routine is called to perform some operation on an external unit, it diverts the request to IO.SYS by issuing int 2Fh function 0802h. IO.SYS reacts just as if its interrupt routine had been called for a unit defined during IO.SYS initialisation, instead of a unit added later by DRIVER.SYS. Note that the intention is not to bypass the DOS kernel, which will ordinarily have taken all the usual steps (like issuing critical section signals) before calling the DRIVER.SYS interrupt routine.

Table 16-11: Continued

External Driver Function 03h—Get Address of First Unit Table

INPUT:

AX	0803h

OUTPUT:

ds:di	address of first Unit Table

This function was introduced with DOS 4. Before, no means was provided to locate the start of the Unit Table chain. Note also that the drive table structure is different in earlier versions.

Since at least DOS 3.20, however, the very first word of IO.SYS (i.e., the word at 0070:0000) has been set during IO.SYS initialization to an address just a little before the first Unit Table for a hard disk. In DOS 3.20 specifically, the int 24h handler in

COMMAND.COM relies on this address to be left at 0070:0000 so that it may consult the IO.SYS Unit Tables and ultimately identify an offending hard disk sector at user level.

Finding the BIOS Drive Number

To a large extent, the Generic IOCTL interface (the eventual understanding of which is the prime motivation for this lengthy digression on drives and disks) removes the need for access at int 13h level. It will be seen that Generic IOCTL services allow the storage device to be handled in terms of physical parameters like head and cylinder number, just like int 13h, but with the convenience of also embodying whatever "spin" that the device driver puts on the way that DOS access the storage.

Still, int 13h access is sometimes necessary, and one difficulty that faces any program working through int 13h (or indeed, communicating directly with the hardware) is that DOS and its users think of their disk storage in terms of drive letters that bear no obvious relationship to the drive number that is needed when accessing the underlying disk through int 13h.

How to reduce the DOS drive letter seen at user level to the BIOS int 13h drive number? Must one search a chain of partition tables to discover whether the user's drive E belongs to the first, second, or third physical hard disk? If drive E is a floppy disk drive, is it out of sequence because it is genuinely a third floppy drive, or has DRIVER.SYS created a logical floppy drive?

The following module shows two ways to determine the int 13h drive number corresponding to a DOS drive letter. The `UseIOSYSDriveTables` function uses the int 2Fh function 0803h (Get Address of First Unit Table) service mentioned above. Alternatively (`#ifdef _NO_IOSYS_`), the `UseInt13Trap` function (not surprisingly) hooks int 13h to see what DOS drive access "turns into" at the int 13h level. As the comments in BIOSDRV.C indicate, however, this second method is more complicated than it at first sounds, and using the IO.SYS Unit Tables is far preferable.

```
/*  biosdrv.c */

/*  This file demonstrates two methods to determine the int 13h drive number
    corresponding to a DOS drive (0-based throughout the file). */

#include    <bios.h>
#include    <dos.h>
#include    <stdlib.h>

#include    <intpack.h>
#include    <iosys.h>
#include    <standard.h>

/*  In this first part of the file, simply select the method.
```

```
    The first, which is available only with recent DOS versions, uncovers
    the drive number used at int 13h level only if the DOS drive is managed
    by the default block device driver in IO.SYS.

    The second method installs a temporary int 13h handler so that the drive
    number used for int 13h access can be recorded when a sector is read
    from the DOS drive. It uses only documented APIs but has all the usual
    disadvantages of invasive testing, made most noticeable in this case by
    the need to "hit the disk". */

static BYTE UseIOSYSDriveTables (WORD);         // first method
static BYTE UseInt13hTrap (WORD);               // second method

BYTE GetBIOSDrive (WORD dosdrive)
{
    #ifdef _NO_IOSYS_                           // for testing purposes, compile
    return (UseInt13hTrap (dosdrive));          // with the _NO_IOSYS_ symbol to
    #else                                       // force use of int 13h trap

    BYTE biosdrive;

    /*  If the first method doesn't work, try the second. */

    biosdrive = UseIOSYSDriveTables (dosdrive);
    if (biosdrive == 0xFF) biosdrive = UseInt13hTrap (dosdrive);
    return (biosdrive);

    #endif
}

/*  ----------------------------------------------------------------------  */

/*  First method. This uses an undocumented int 2Fh function (available
    only since DOS 4) to obtain a pointer to the chain of tables in which
    IO.SYS maintains its information about the DOS drives supported by the
    default block device driver. */

static struct IOSYSDRIVETABLE _far *GetIOSYSDriveTable (void);

static BYTE UseIOSYSDriveTables (WORD dosdrive)
{
    struct IOSYSDRIVETABLE _far *ptr;

    /*  Get the pointer. */
    ptr = GetIOSYSDriveTable ();
    /*    Follow the chain (each table contains a far pointer to another—not
          necessarily in ascending drive order) to the table for the specified
          DOS drive number. Another field in the table gives the BIOS drive
          number that is used for int 13h access. */

    while ((WORD) ptr != 0xFFFF) {
          if (ptr -> cDOSDrive == (BYTE) dosdrive)
             return (ptr -> cBIOSDrive);
          ptr = ptr -> lpNext;
    }

    /*    If the end of the chain is reached without matching the DOS drive
          number, then the drive is not managed by the default block device
          driver in IO.SYS and the int 13h drive number remains unknown. */

    return (0xFF);
}

static struct IOSYSDRIVETABLE _far *GetIOSYSDriveTable (void)
{
    if (_osmajor < 4) {
          bad:
          return ((struct IOSYSDRIVETABLE _far *) -1);
    }
    _asm {
             mov     ax,0800h
             int     2Fh
             cmp     al,0FFh
             jnz     bad

             push    ds
             mov     ax,0803h
             int     2Fh
             mov     ax,di
             mov     dx,ds
             pop     ds
    }
}

/*  -------------------------------------------------------------------------  */

/*  Second method. This generates a sector read from DOS level, having
    first inserted a small piece of code in the int 13h chain to observe
    whether or not int 13h is used to control the drive hardware. */
```

```
static void _interrupt _far int13h (struct INTPACK);
static void (_interrupt _far *lpOrgInt13h) (void);

static BYTE buffer [0x0200];
static BYTE int13h_dl;
static BOOL int13h_read;

extern void _dos_getbreakflag (BYTE *);
extern void _dos_setbreakflag (BYTE);
static BYTE OrgBreakFlag;

static struct DISKIO {
    DWORD dwStart;
    WORD wCount;
    void _far *lpBuffer;
} diskio;

static BYTE UseInt13hTrap (WORD dosdrive)
{
    /*  Disable Ctrl-C checking to prevent termination in the unlikely
        event that a break is pending when the _dos_setvect function is
        called to unhook the int 13h probe. */

    _dos_getbreakflag (&OrgBreakFlag);
    if (OrgBreakFlag) _dos_setbreakflag (OFF);

    /*  Install this function's int 13h hook, remembering to initialize the
        variable in which the hook routine is meant to record the value of
        register dl (the drive number) when int 13h is used. */

    lpOrgInt13h = _dos_getvect (0x13);
    _dos_setvect (0x13, int13h);
    int13h_dl = 0xFF;
    int13h_read = FALSE;

    /*  Trigger an int 13h call by attempting to read a sector (logical
        sector 0) from the DOS drive. The int 25h facility is the lowest
        level service that works with DOS drive definitions. In a simple
        world, int 25h does little more than verify that the drive is valid
        at DOS level, then call the responsible device driver. The success
        or failure of the read attempt is immaterial.

        Of course, int 25h may be intercepted before it reaches the DOS
        kernel. Whether or not this produces multiple int 13h calls,
        possibly even to different drives, is for the int 13h probe to deal
```

```
     with. One thing that may be reasonably well assumed is that no
     int 24h exceptions can be generated (which simplifies the defence to
     inhibiting breaks, as done above). */

diskio.dwStart = 0;               // int 25h may need arguments in a
diskio.wCount = 1;                // parameter block (typically for
diskio.lpBuffer = buffer;         // DOS 4 and higher)

_asm {
         push    ds               // int 25h doesn't preserve
         push    bp               // registers in earlier DOS versions
         push    di
         push    si

         cmp     _osmajor,4       // DOS 4 and higher are certain to
         jnb     newstyle         // admit input as parameter block

         mov     bx,offset buffer // For earlier DOS versions, try
         mov     cx,1             // the register style of input
         xor     dx,dx
         mov     al,byte ptr dosdrive
         int     25h
         pop     bx               // remove interrupt flags from stack
         jnc     done

         cmp     ax,0207h         // error 0207h indicates input should
         jnz     done             // be presented in parameter block

     newstyle:
         mov     bx,offset diskio
         mov     cx,0FFFFh
         mov     al,byte ptr dosdrive
         int     25h

         popf                     // int 25h always leaves flags
     done:
         pop     si
         pop     di
         pop     bp
         pop     ds
}

/*   Detach the int 13h probe. This is the DOS call that could terminate
     the program preemptively were break-checking not disabled.
     Remember that when BREAK=ON, DOS checks the keyboard device (the
          most recently-loaded character device driver that had bit 0 set in
          its attribute word) and the stdin handle for a Ctrl-C character
          before proceeding with the body of the int 21h call, even a
          seemingly "simple" call like the 21/25 made by _dos_setvect. */

    _dos_setvect (0x13, lpOrgInt13h);

    /*    Now that the int 13h probe is disconnected, return Ctrl-C processing
          to its previous condition. */

    if (OrgBreakFlag) _dos_setbreakflag (OrgBreakFlag);
    return (int13h_dl);
}

static void _interrupt _far int13h (struct INTPACK r)
{
    /*    There is no telling how much int 13h activity will be generated by
          the device driver. Worse, the test is exposed to interruption and a
          TSR may attempt some disk operation. Worse still, int 25h may have
          been trapped by a cautious disk cache that flushes possibly huge
          amounts of material to possibly many different disk drives.

          A read request is expected and the test cannot be thought conclusive
          if none is trapped (at least, not without imposing assumptions about
          the sequence of int 13h operations that a device driver might use to
          prepare for reading—and even then, the possibility that int 25h
          has been hooked means that such preparations might be for some drive
          other than the one whose BIOS-level identity is sought).

          The possibility exists that more than one read request may be seen.
          If this happens, the resulting confusion must be untangled, but
          nothing about the expected read request is sure; it cannot be
          supposed that logical sector 0 translates to physical sector 1,
          head 0, cylinder 0 and it cannot even be thought that the device
          driver will ask for only one sector (it may be caching consecutive
          sectors internally).

          The best that can be arranged is a defence against multiple reads
          (from different drives) occurring while the int 13h probe is
          installed. */

    if (r.ah == _DISK_READ) {
          if (NOT int13h_read) {
              int13h_read = TRUE;
              int13h_dl = r.dl;
```

```
        }
        else if (int13h_dl != 0xFF) {
            if (r.dl != int13h_dl) int13h_dl = 0xFF;
        }
    }

    /*  All requests must be passed onwards since it is unclear which ones
        the probe would be entitled to fail. This has the unfortunate
        consequence of "hitting the disk" even when the sequence from
        int 25h to int 13h encounters no interference. */

    _chain_intr (lpOrgInt13h);
}
```

BIOSDRV.C will be used later to build the larger DRVINFO program. In the meantime, the `GetBIOSDrive` function can be called from a simple test rig, such as the following:

```
main()
{
   unsigned dos_dr, bios_dr;
   for (dos_dr = 0; dos_dr < 26; dos_dr ++)
      if ((bios_dr = GetBIOSDrive(dos_dr)) != 0xFF)
         printf("%c\t%02Xh\n", 'A' + dos_dr, bios_dr);
}
```

The program needs to be linked with BREAK.C in the LIB subdirectory on the accompanying disk. The program can be compiled with `-D_NO_IOSYS_` to use the int 13h hook rather than int 2Fh function 0803h; indeed, the purpose of the int 13h hook is simply to show the sort of desperate (and necessarily flawed) measures required to solve the DOS/BIOS drive mapping problem before the availability of int 2Fh function 0803h in DOS 4 and higher. In any case, the resulting BIOSDRV test will of course display something like the following (in this example, **device=device.sys /d:0** has made drives A and E share the same physical floppy drive):

```
A          00h
B          01h
C          80h
E          00h
I          80h
```

Having seen the IO.SYS drive table, and used it to determine the DOS/BIOS drive mapping (without the mess associated with hooking int 13h), we now turn to the higher-level structures maintained by DOS.

From Device Driver to File System

At kernel level a disk should be an array of sectors, ready for organization into more abstract structures that support a file system. The native DOS file system is based on a File Allocation Table (FAT) that records the distribution on disk of some (possibly) disjoint collection of allocation units (or clusters) that the file system must present to applications as a sequential whole—a file.

When data is read from a file, the DOS kernel consults whatever structures it deems relevant and eventually decides where on the disk it will find the file data. But the language in which it expresses the concept of location is one in which the storage device is an array of logical sectors, starting from zero and working upward. It is entirely up to the device driver to translate this abstract view into something more meaningful for accessing the hardware (for instance, into a language of cylinders and heads).

The DPB

Each unit declared by each block device driver installed for the session becomes a DOS drive. This relationship between device unit and DOS drive is embodied in a Drive Parameter Block (DPB). This DPB is the DOS kernel's complete picture of what it regards as a physical drive. Note, however, that from the perspective of the device driver, DOS's physical drive may be a logical unit: such is the relativity of these things. Newly-documented int 21h function 32h returns the DPB corresponding to a given drive.

Table 16-12: Drive Parameter Block (DPB)

OFFSET	SIZE	DESCRIPTION
00h	byte	0-based DOS drive number
01h	byte	Unit number on block device driver responsible for access to drive
02h	word	Bytes per sector
04h	byte	Sectors per cluster minus one. Because the number of sectors per cluster is necessarily a power of two, a bitwise AND with this field is equivalent to finding the remaining sectors after grouping sectors into clusters.
05h	byte	Logarithm base two of sectors per cluster. A bitwise right shift by the count given in this field is equivalent to finding the number of whole clusters in a given number of sectors.
06h	word	Logical sector number at which first FAT begins
08h	byte	Number of FAT copies
09h	word	Number of directory entries for which the root directory has space. When the DOS kernel calculates the number of sectors required for the root directory, any excess entries over a multiple of 16 are rounded up, but the default block device driver in IO.SYS rounds down.

Table 16-12: Continued

OFFSET	SIZE	DESCRIPTION
0Bh	word	Logical sector number at which file data begins
0Dh	word	Highest valid cluster number. The first data cluster is numbered 2, so this field is also the number of clusters plus one. If this field is < 0FF6h, then DOS interprets the FAT in terms of 12-bit entries; otherwise, FAT entries are supposed to be 16-bit.
0Fh	word	Sectors per FAT
11h	word	Logical sector number at which root directory begins
13h	dword	Address of device driver responsible for access to drive
17h	byte	Media descriptor
18h	byte	Set to FFh to force a BPB rebuild after the next Media Check
19h	dword	Address of next DPB in chain (low word is FFFFh if this is the last DPB in chain)
1Dh	word	Cluster number allocated most recently (0000h if the number is not known). DOS allocates clusters from the start of the disk's data area upward.
1Fh	word	Number of clusters remaining unallocated (FFFFh if the number is not known)

The structure is shown here in the form that applies for DOS 4 and higher. In earlier versions, only one byte is allowed for the field at offset 0Fh which gives the number of sectors per FAT.

The DPB has no place for characteristics like sectors per track that would obviously describe hardware. Even the number of bytes per sector given at offset 02h is not a property of the hardware, since at this level, the term "sector" refers just to the block in which data is transferred between the DOS kernel and the responsible device driver: there is no reason why this must be the same size as the sector used for transferring data between the device driver and the hardware.

Not surprisingly, the DPB is an expression of the DOS kernel's interests. The disk is treated as an array of sectors and some of the DPB fields simply serve as markers in this array to show where one region ends and another starts. Given a logical sector number x, the simple comparisons shown following allow the DOS kernel to determine very quickly what role is played by the sector.

As far as the DOS kernel is concerned, the reserved sectors are a property of the responsible device driver. No code in the DOS kernel makes any assumption about the contents of this area at the start of the disk.

Table 16-13: Identification of Sector Type

x < DPB [06h]	reserved sector
DPB [06h] ≤ x < DPB [11h]	FAT sector
DPB [11h] ≤ x < DPB [0Bh]	root directory sector
DPB [0Bh] ≤ x	data sector

Immediately after this region that is private to the device driver, two areas are set aside for maintenance of the file system. As a safety provision against device errors (typically media defects), the disk may contain multiple copies of the FAT. The DPB records the number of duplicates and the size (in sectors) of each. No code in the DOS kernel cares how many FAT duplicates the disk supports. The first copy is of course, the most important, but when the kernel updates a FAT sector, it writes the same sector DPB [08h] times, DPB [0Fh] sectors apart. Each write must be successful. For read operations, the kernel is happy to accept whichever copy of the desired FAT sector the device driver is able to read without error.

The largest part of the disk is available for storing data in files. DOS manages this region in allocation units called clusters. Here also, information is kept in a form that is immediately convenient for common operations. This is most evident with the fields at DPB offsets 04h and 05h, which the kernel uses to its slight advantage when converting between sectors and clusters.

To each cluster in the data area there corresponds a FAT entry. If the cluster belongs to a file, then the value of its FAT entry is the index of the next cluster in that file. The special value 0000h indicates that a cluster is not allocated. The special value 0001h is used temporarily while building cluster chains and can be seen on disk only by interrupting an allocation operation. The first cluster in the data area is number 0002h. The last cluster of data on the drive has the index DPB [0Dh]. A higher FAT entry must either be invalid (which triggers an int 24h exception to warn of danger) or be one of the values reserved as marking the end of a cluster chain. For a drive using 12-bit FAT entries, these end-of-file markers are 0FF0h and all values ≥ 0FF8h. In the 16-bit case, FAT entries ≥ FFF8h mark the end of a chain.

It is important to discuss how DOS knows the size of the FAT entry used by a disk. The FAT-size flag, as it were, is whether the field at offset 0Dh in the DPB is less than 0FF6h; if so, then the FAT is interpreted in terms of 12-bit entries and if not, then 16-bit entries are assumed. Note that DPB [0Dh] is the highest valid cluster number for the drive. It is not the number of clusters, but rather, the number of clusters plus one. This means that strictly speaking, the official documentation is wrong to assert that DOS uses 12-bit FAT entries for drives with 4086 or fewer clusters. Although the exact position of the changeover from 12-bit to 16-bit FAT is probably without practical consequences, it is curious that the DOS kernel, the official documentation and the latest versions of such utilities as FORMAT and CHKDSK are able to produce three different answers.

The question remains of how the DOS kernel obtains the information it keeps in the DPB. The device driver responsible for the unit knows very little, if anything, of the kernel's plans for the disk. The device driver understands that all the kernel's requests to read or write sectors on the disk will be phrased in terms of logical sectors: it is for the device driver to translate the logical sector numbers into parameters appropriate to the hardware.

In addition to this, there is some translation that works in the opposite direction. In the earlier discussion of the default block device driver, it was observed that the boot sector contains a BPB. It was also observed that this is not strictly required. The BPB is just the structure in which the DOS kernel requires device drivers to present their description of a device unit. The description is expected when block device drivers are initialized and also whenever the DOS kernel has cause to call the Build BPB function of the device driver interface (typically after the Media Check function has returned a negative result). Note that the device driver is not obliged to work with a BPB for its own purposes, much less to store a BPB on disk. Indeed, it has been seen that even for standard disk formats where there is a BPB in the boot sector, the BPB used by the default block device driver may differ significantly from the version on disk.

Given a pointer to a BPB, undocumented int 21h function 53h updates some fields in the DPB whose address is also passed to the function:

Table 16-14: DOS Function 53h—Update DPB from BPB

INPUT:

AH	53h
DS:SI	Address of BPB from which to take data
ES:BP	Address of DPB in which to enter new values

OUTPUT:

al	corrupt

The function's intended use is to update a DPB with information from a freshly-obtained BPB. The following fields are unrelated to this purpose and are left untouched:

```
DPB [00h]   DOS drive number
DPB [01h]   device driver unit number
DPB [13h]   address of responsible device driver
DPB [18h]   media check flag
DPB [19h]   address of next DPB
```

Two other fields also unrelated to the main purpose are initialized:

```
DPB [1Dh]   cluster number allocated most recently is set to 0000h (unknown)
DPB [1Fh]   number of free clusters is set to FFFFh (unknown)
```

DOS 5 tries to check that the declared size of the FAT is large enough to support the number of clusters deduced from the total disk capacity (that is, after allowing space for reserved sectors, the FATs, and the root directory). The intention is that if the FAT is too small, then DOS 5 should recognize only as many clusters as can be described by that size of FAT.

From DOS 4 onward, a special case is made if the BPB gives the number of FAT copies as zero. The only DPB fields taken from the BPB are those representing bytes per sector and the number of FAT copies. The fields at offsets 1Dh and 1Fh (related to cluster allocation) are initialized as usual. Peculiarly, the value entered as the highest valid cluster number is the number of bytes per sector plus 1.

Coordinating Directory Trees Across All Drives

In addition to the physical DOS drives, there may be virtual DOS drives supported through an array of Current Directory Structures. The CDS provides a layer of indirection with DOS drive letters and, as its name suggests, is also DOS's "bookmark" (so to speak) in the drive's directory tree.

Begin with the very simplest case of a CDS for a local, physical DOS drive. DOS functions that refer to the drive cause the kernel to look first in the CDS, where it finds a pointer to the DPB, and then to the device driver responsible for accessing the physical storage. If the DOS function also refers to a path or pathname, then DOS may have to follow the path from the root, looking into each subdirectory on the way. But before it does this, it checks whether the path is a continuation of the one at offset 00h in the CDS. This path is the current directory for the drive. It is quite likely that DOS will already have found this path within the file system and have recorded the subdirectory's location (at offset 49h in the CDS; see Table 16-15). Figuratively speaking, it is as if someone were keeping their thumb to mark a place in a dictionary. The practical result can be seen in significantly better access times when programs change directory.

Even a cursory glance at the following description will show that the CDS supports much more than this notion of a current directory. Most importantly, when DOS consults the CDS for a given drive, it need not be directed to a DPB. The DOS kernel contains code only for supporting its native FAT system; the CDS is the signpost to indicate that other software takes responsibility for the drive concerned. Instead of being handled by a device driver, operations for that drive are passed to a "redirector," the defining example of which is the REDIR.EXE program for Microsoft-compatible networks (MSCDEX.EXE for CD-ROM drives is another example, but it is hardly a typical redirector, and certainly not the defining example as is sometimes claimed). Because of this association, the drive may be thought of as being remote, but the principle is that the redirector acts as a kernel extension to support an alien file system.

Between the extremes of local and remote drives lie the more simple indirections most commonly established with the SUBST and JOIN utilities. Though it is unlikely that anyone actually adopts the following as a working practice, the idea behind JOIN is to eliminate all mention of drives by making them seem like directories (e.g., **join a:**

c:\floppy1). Drive substitution is an altogether much more useful concept since its effect is to allow a user to maintain more than one current directory on a physical DOS drive.

The DOS internal SysVars structure whose address is returned by int 21h function 52h (see Part I) includes a pointer to the DOS CDS array (see `GetCDS` and `GetSysVarsAddress` in DOSLEVEL.C below).

Table 16-15: The Current Directory Structure (CDS)

OFFSET	SIZE	DESCRIPTION	
00h	43h	null-terminated path to current directory; or for drive JOINed to a directory, null-terminated resolved path to joint	
43h	word	flags	
		0080h	remote drive hidden from the redirector's Assign-List and not subject to the functions that make or break network connections—set for CD-ROM drives (MSCDEX)
		1000h	drive SUBSTitutes for a directory of some other drive—specifically, for the root component of the path given at offset 00h
		2000h	drive is JOINed into the directory tree of some other drive—at the point indicated by the path given at offset 00h
		4000h	drive is valid
		8000h	drive is remote
45h	dword	for local drive: address of DPB	
49h	word	for local drive: cluster containing first sector of current directory; or 0000h if current directory is the root; or FFFFh if starting cluster is unknown	
	dword	for remote drive: address of redirector block in DOS 5, but available for use by IFS driver in DOS 4	
4Dh	word	for remote drive: user value	
4Fh	word	number of characters in root component of current directory given at offset 00h	
51h	byte	for remote drive in DOS 4 only: device type	
52h	dword	for remote drive: address of redirector block in DOS 5 or of IFS driver in DOS 4	
56h	word	for remote drive in DOS 4 only: available for use by IFS driver	

The fields from offset 51h on are used in DOS 4 to support installable file systems. In earlier DOS versions, these extra fields did not exist.

In DOS 4, the device type 04h denotes a network drive. Other device types are allowed but the redirection appears in the Assign-List only when retrieved with the undocumented int 21h function 5F06h. Connections can be made and broken to such devices using DOS functions 5F03h and 5F04h, but only through an extended parameter convention. To some extent the DOS 5 redirector offers a similar facility via bit 7 in the CDS flags.

The user value at offset 4Dh is provided to help manage drive redirections by associating each assignment with a word of the DOS caller's choice. In DOS 4 the field at offset 56h is similarly available for the convenience of the installable file system.

Observe that the field at offset 49h serves different purposes, depending on whether the drive is local or remote. If the drive is local, only one word is required. The second word is initialized to FFFFh when the CDS is first built by SYSINIT but will be zero if the CDS is later rebuilt as a consequence (for instance) of pausing drive redirection with DOS function 5F01h. For remote drives the field is available for the redirector's own use.

The DOSLEVEL.C module given below shows various ways to inspect the CDS and DPB in order to extract information about a drive's status in the overall file system. Together with the BIOSDRV.C module, it anticipates the development of a small DRVINFO utility.

Because the CDS has a DPB field for local drives, a program can use the CDS to get the DPB for a given drive, without "hitting the disk" as int 21h function 32h does. Another way to get a DPB without hitting the disk is to walk the DPB chain (to which SysVars contains a pointer). However, pulling a DPB out of a drive's CDS entry is far preferable. DOSLEVEL.C shows code to walk the DPB chain purely for reference, within `#ifdef FALSE`. DOS itself almost always uses the CDS method rather than walk the DPB chain.

```
/* doslevel.c */

#include    <stdlib.h>
#include    <string.h>

#include    <cds.h>
#include    <dpb.h>
#include    <sysvars.h>

/*  Two methods are given for finding a DPB, given a 0-based drive number.
    The first method uses a documented int 21h function, but "hits the
    disk", which is generally a disadvantage if only the drive properties
    are being investigated. Worse, the function is unsuccessful if the disk
    drive is empty. In essence, the second method follows the steps taken
    by the DOS kernel in the first method. It avoids disk access, but is
    susceptible to future changes in the CDS structure (as happened with
    version 4). */

#ifdef _USE_DOS_API_
```

```
struct DPB _far *GetDPB (WORD drive)
{
    /*  For any valid local drive, int 21h function 32h returns with ds:bx
    addressing the DPB for the corresponding "physical" DOS drive. It
    is an old DOS function that indicates failure by al = FFh rather
    than a set carry flag. */

    _asm {
        push    ds
        mov     dx,drive
        inc     dx  // DOS function requires 1-based drive number
        mov     ah,32h
        int     21h
        mov     dx,ds
        pop     ds
        cmp     al,0FFh
        jz      bad
        mov     ax,bx
        jmp     done
    bad:
        xor     ax,ax   // return NULL if no DPB exists for drive
        cwd
    done:
    }
}

#else

static struct SYSVARS _far *GetSysvarsAddress (void);
static struct CDS _far *GetCDS (WORD);

/*  The objective of the function GetDPB is to find the DPB for the given
    0-based drive by taking the same steps as the DOS kernel for int 21h
    function 32h, but without having to "hit the disk". */

struct DPB _far *GetDPB (WORD drive)
{
    struct CDS _far *cdsptr;

    /*  Begin by finding the CDS for the drive and checking that the drive
    is valid and local. */

    cdsptr = GetCDS (drive);
    if (cdsptr == NULL) return (NULL);
    if (NOT (cdsptr -> wFlags & CDS_VALID)) return (NULL);
    if (cdsptr -> wFlags & CDS_REMOTE) return (NULL);

    /*  Read the DPB address from the CDS. This method - used by the DOS
    kernel for almost all its drive dereferences - accounts for
    indirection at directory level and allows a DPB to be found for
    SUBSTed drives. */

    return (cdsptr -> lpDPB);
}

static struct SYSVARS _far *GetSysvarsAddress (void)
{
    _asm {
        mov     ah,52h
        int     21h
        mov     ax,bx
        mov     dx,es
    }
}

static struct CDS _far *GetCDS (WORD drive)
{
    struct SYSVARS _far *sysvarsptr;
    struct CDS3 _far *cds3ptr;

    sysvarsptr = GetSysvarsAddress ();
    if ((BYTE) drive >= sysvarsptr -> cLogicalDrives)
        return (NULL);

    /*  Use the drive number to index the array of CDSs. Allow for the
    change in structure that accompanied DOS 4. */

    if (_osmajor >= 4)
        return (sysvarsptr -> lpFirstCDS + drive);
    else {
        cds3ptr = (struct CDS3 _far *) sysvarsptr -> lpFirstCDS + drive;
        return ((struct CDS _far *) cds3ptr);
    }
}

#if FALSE

/*  An alternative way to find the DPB for a drive - but only if the drive
    corresponds directly to a device unit - is to follow the DPB chain.
    This is included here for reference only. DOS itself does not use this
    method, except in a few cases where it is interested only in "physical"
    drives. */

static struct DPB _far *GetRawDPB (WORD drive)
{
    struct DPB _far *ptr = GetSysvarsAddress () -> lpFirstDPB;

    while ((WORD) ptr != 0xFFFF) {
        if (ptr -> cDrive == (BYTE) drive)
            return (ptr);
        ptr = ptr -> lpNext;
    }
    return (NULL);
}

#endif
#endif

/* ======================================================================== */

/*  Similarly, two methods are provided for analysing properties lent to a
    drive via the CDS. Again, using the API (undocumented in this case)
    triggers a disk access.

    For both methods, pass the results as a set of printf style arguments
    ready for output via a function called PutComment. This lets an
    external module ensure consistent formatting. */

extern void PutComment (CHAR *, ...);

static CHAR pathname [0x80];

#ifndef _USE_DOS_API_

void DescribeCDSProperties (WORD drive)
{
    struct CDS _far *cdsptr;
    register WORD count;
    register CHAR *ptr;

    cdsptr = GetCDS (drive);
    if (cdsptr == NULL OR NOT (cdsptr -> wFlags & CDS_VALID)) return;

    if (cdsptr -> wFlags & (CDS_SUBST | CDS_REMOTE)) {
        count = cdsptr -> wRootCount;
        _fmemcpy (pathname, cdsptr -> cDirectory, count);
        ptr = pathname + count;
        if (count == 2) *ptr ++ = '\\';
        *ptr = '\0';
        PutComment ("%s %s",
            cdsptr -> wFlags & CDS_SUBST ? "substituted for" : "mapped to",
            pathname);
    }

    if (cdsptr -> wFlags & (CDS_JOIN))
        PutComment ("joined to %Fs", cdsptr -> cDirectory);
}

#else

static WORD _dos_canonicalise (CHAR *, CHAR *);

#define NULCHARS    (sizeof ("\\NUL") - 1)

#pragma pack (1)
static struct {
    CHAR drive;
    CHAR colon;
    CHAR backslash_nul [NULCHARS + 1];
} test_spec = {'X', ':', "\\NUL"};
#pragma pack ()

void DescribeCDSProperties (WORD drive)
{
    WORD parsedlength;

    /*  One way to see if a drive is mapped to another drive's directory
    tree is to use int 21h function 60h (undocumented, but indispensable
    to many standard DOS utilities and raised to the level of the DOS
    command line as TRUENAME) to resolve a drive's root directory
    (letter, colon and backslash). However, this will ordinarily "hit
    the disk".

    An exceptional case may be used to advantage. Instead of parsing
    just the drivespec and leading backslash, parse a device name in the
    root directory. This will cause DOS to consult the CDS to validate
    the drive, but shortly afterwards, DOS will exit from its parsing
    routines when it discovers that the "file" is really just a
    character device. This complicates the programming a little, since
    the name of the character device has to be removed from the parsed
    result before the drive translation can be reported - but it avoids
    "hitting the disk".

    One detraction that seems unavoidable is that short of using trial
    and error on all the subdirectories of every drive on the system,
    int 21h function 60h will not reveal the directory to which a drive
    has been JOINed. */

    test_spec.drive = (BYTE) (drive + 'A');
    if (_dos_canonicalise (pathname, (CHAR *) &test_spec) != 0) return;

    /*  If the input path changes when canonicalised, then the drive is
    mapped elsewhere in the file system. Of course, the \NUL component
    at the end of the path must be removed in order to find out what the
    bare drivespec translates to. */

    if (memcmp (pathname, (CHAR *) &test_spec, sizeof (test_spec)) != 0) {
        parsedlength = strlen (pathname);
        if (parsedlength > NULCHARS) {
            if (memcmp (pathname + parsedlength - NULCHARS,
                test_spec.backslash_nul, NULCHARS) == 0) {
            *(pathname + parsedlength - NULCHARS) = '\0';
            PutComment ("translates to %s", pathname);
            }
        }
    }
}

static WORD _dos_canonicalise (CHAR *outptr, CHAR *inptr)
{
    _asm {
        mov     si,inptr
        push    ds
        pop     es
        mov     di,outptr
        mov     ah,60h
        int     21h
        jc      done
        xor     ax,ax
    done:
    }
}

#endif
```

The issue of an API "hitting the disk" comes up a second time in DOSLEVEL.C, in the `DescribeCDSProperties` function. Here the choice is either to use an API (albeit undocumented) that hits the disk, int 21h function 60h (TRUENAME), or to avoid the disk access by directly manipulating the CDS in memory. This function will, once again, be used later on in the DRVINFO program.

Logical Drive Mapping

There is a subject area in DOS—logical drive mapping—that ought to be treated very carefully by *every* program contemplating operations with floppy disks. Instead the usual approach is to avoid the subject entirely. This is true even of packages that specialize in providing disk services—including some with a large popular following among users and programmers alike. Whether the problem arises out of genuine ignorance or a general desire for it to go away is of arguably less importance than the consequences of its evasion. Some programs, such as Microsoft Windows, can be made to crash as the penalty for not getting logical drive mapping right.

Most programmers and many users are familiar with the existence of a floppy drive B on machines fitted with only one floppy drive. DOS creates a "logical" floppy, ostensibly to support file transfers from one floppy disk to another. With only one physical device to service two disks, it is helpful to imagine that two drives are available—one for each disk. Ordinarily it is left to the system to keep track of the "active" drive—the one in control of the physical device. If asked to access a logical unit that is not the one currently using its shared physical device, the system (typically the default block device driver in IO.SYS) must alert the user. For example, if drive A has just been used to read a floppy disk and IO.SYS receives a request to write to drive B, it directs the user to

```
Insert the disk for drive B: and press any key when ready
```

Thereafter, logical drive B is in control of the physical device and the user will be prompted again only if the disk for logical drive A is needed. Note the prompt is *not* (as frequently and unthinkingly claimed) a critical error message about the drive being empty. It is issued from a much deeper level and indicates only that the physical drive is changing its logical association to a different drive letter.

This arrangement for drives A and B is configured at startup if the BIOS reports the presence of only one floppy disk drive. With two or more floppy drives there is no immediate need to create logical partners, so deliberate action after startup is required if the user wants to assign more than one letter to an existing physical floppy drive. The DOS manual presents clear instructions in several places, so the facility cannot be dismissed as wildly esoteric and programmers have little excuse for not accommodating it. As explained in the MS-DOS *User's Guide* (but unfortunately ignored by many DOS programmers), DRIVER.SYS can not only add additional floppy drives to the system, but (by using the **/d** switch to specify the number of an existing drive) assign extra logical drive letters to a single physical drive. Too often, programs simply assume that drive A/B is the only possibility for two logical drives to share the same physical drive.

Programs that manipulate disks in two or more drives must normally be aware that the different DOS drives may in fact refer to the same physical device. As an example, consider the command

```
diskcopy a: e:
```

If the two drives are genuinely independent, then both disks involved may be inserted when the program starts and the data transfer may proceed without troubling the user any further. If, however, DRIVER.SYS has made drives A and E share a single physical drive, then the user must be asked from time to time to insert the correct disk—source or target—just as if the original command had been

```
diskcopy a: a:
```

Shells and diagnostics utilities that try to describe the available drives are also affected by the possibility of floppy drives sharing the one device. For many such programs, investigation of drive properties requires some form of disk access. In some cases, describing any disks actually present in the drives is also a goal, so each drive is tested deliberately for a readable disk. This causes no problem if each floppy drive letter corresponds to its own physical device. At worst, the user may have to wait a little for a slow drive to signal a time-out: almost all these programs know how to protect their users from critical error messages.

If, however, one of the drives is a currently inactive logical construction, then "hitting the disk," even inadvertently, will trigger the system's disk swap prompt, annoy the user, and almost certainly make a mess of the program's carefully prepared display. As some measure of this subject's relegation to the programming wilderness, note that the MSD utility distributed for diagnostic purposes with Windows and other Microsoft packages is among those that seem totally unaware of logical drive mapping.

Microsoft Windows itself suffers very badly from problems with logical drive mapping. For example, continue with the example where DRIVER.SYS has made DOS drives A and E share the same physical floppy disk drive, and suppose that drive A is currently active. Imagine also that a disk containing an executable program is in the drive. Start Windows in enhanced mode. For reasons explained below in the discussion of the `GetDriveType function`, most Windows applications (including the Program Manager) deem the inactive drive E to be non-existent; asking PROGMAN to run the program by giving it a path on E: will not trigger a change in the logical drive map (since PROGMAN simply ignores that drive and looks elsewhere for the executable). Instead, open a DOS Prompt and run **dir e:** to deactivate the drive that PROGMAN does believe exists, then switch back to PROGMAN and attempt to run the program from the floppy disk, giving its path on drive A (no longer active). Windows hangs.

To help programmers with this sort of logical drive mapping problem, DOS offers two functions at int 21h level. Get Logical Drive Map (int 21h function 440Eh) may reveal that a given DOS drive shares its physical storage with some other DOS drive or drives. If so, the function may either confirm that the given drive controls the shared storage device or show which of its partners is active. The other function—Set Logical Drive Map (int 21h function 440Fh)—directs the responsible device driver to regard the given drive as active, thereby avoiding the annoying "Insert the disk" prompt.

Table 16-16: DOS functions 440Eh and 440Fh—Get/Set Logical Drive Map

INPUT:

AX	440Eh or 440Fh respectively for Get/Set Logical Drive Map
BL	1-based DOS drive; or 00h to denote default

SUCCESS:

cf	clear
al	1-based DOS drive number in control of physical device shared with drive in BL; or 00h if drive in BL does not share its physical device with other drives
ah	corrupt

FAILURE:

cf	set	
ax	0001h	if drive BL is remote or if device driver does not support logical drive mapping
	0005h	if device driver returns an error
	000Fh	if drive BL is invalid

The function frames a device driver request packet for command 17h (Get Logical Drive) or 18h (Set Logical Drive) as appropriate. When the device driver is called, the field at offset 01h is the unit number corresponding to drive BL. However, the device driver must return a 1-based DOS drive number (or 00h) in this field, *not* a unit number.

The device driver attribute word must have bit 6 set to indicate support for logical drive mapping; otherwise, DOS does not call the driver and returns the Invalid Function error instead.

DOS 4 only allows logical drive mapping for remote drives as long as the installable file system (IFS) driver has bit 11 clear in its attribute word to indicate that the storage device is local. This means that the native FAT file system has been replaced but the DOS kernel still associates the drive with a DPB and device driver. In this case, bit 6 must still be set in the device driver attribute word to indicate support for logical drive mapping.

The particular implementation of Set Logical Drive Map in IO.SYS regards units with fixed media as being always active, even if Get Logical Drive Map reports a unit as shared and inactive. This discrepancy is largely academic since DRIVER.SYS does not support the creation of units that share fixed media.

It may help to understand that each physical disk drive may be assigned several units at device driver level (which ultimately become DOS drive letters). Only one of these units is considered active at any time. Refer to the bit flags in the IO.SYS Drive Table earlier in the chapter to see how IO.SYS keeps track of the connection for drives controlled via int 13h. In the flags word bit 4 is set aside to show that the unit shares its physical device with others and bit 5 marks the unit as active. There is little more to it than that. The Get Logical Drive Map function drops through to the device driver to ask whether the unit is shared and, if so, would the device driver please look in its tables to find the active unit with the same int 13h number.

To summarize, the idea of attaching several letters to one physical drive relies on indicating clearly which letter is currently in use. The device driver must step in with a default message if it has not seen this indication from higher levels, but it is usually preferable for applications to handle the swap beforehand. The tool for this purpose is int 21h function 440Fh (Set Logical Drive Map).

Consider the DISKCOPY example given above. Once the source disk has been read into memory, the program may prompt the user to insert the target diskette, offering some reminder that this is the disk meant to receive data from the source disk and perhaps even asking for confirmation. After doing all this, it should use DOS function 440Fh to tell the device driver that the changeover from A to E has already been supervised at user level; otherwise, the device driver will have no choice but to issue a redundant prompt about inserting a disk.

The default prompt, incidentally, is written to the screen in teletype mode via int 29h. Even applications that do not care about the wording of the prompt may prefer not to have their carefully prepared display disrupted. This is another reason for using Set Logical Drive Map, if only to have the same words appear in a pretty window.

Asynchronous Drive-Swap Notification

The MS-DOS Shell offers an example of a program that arranges its own prompt instead of having the default message intrude on the display. Curiously, it does not anticipate the change in logical mapping as described above and in the official technical reference, but rather waits for IO.SYS to alert it.

In DOS 5 and 6, when IO.SYS issues its disk swap prompt, it advertizes the event in two ways. First, if Windows enhanced mode is running, IO.SYS calls the DOSMGR VxD (as explained in Part 1, IO.SYS calls int 2Fh function 1684h to get the V86 API entry point for device ID 15h, and then calls function 1 in the API). DOSMGR in turn directs all other virtual devices to set the focus to the virtual machine which has triggered the remap. The most obvious effect at user level is to force the DOS session out of a window and into full screen. The virtual machine in question need not be the current one, so the change of focus can be nontrivial. For its own part, DOSMGR accounts for time lost while the system remains in a long critical section waiting for the user to respond. It is not possible to switch away from the disk swap prompt. In earlier DOS versions, which offer Windows no explicit help, notification is arranged by placing a breakpoint at the instruction where IO.SYS waits for the user to press a key (see Part 1 for an explanation of Windows breakpoints).

IO.SYS broadcasts the second advertisement (the one intended for DOSSHELL.EXE and with no parallel implementation in earlier DOS versions) through an undocumented int 2Fh function:

Table 16-17: IO.SYS Broadcast Preceding Logical Floppy Drive Swaps—int 2Fh Function 4A00h

INPUT:

AX	4A00h
CX	0000h
DL	0-based DOS drive number being activated

OUTPUT:

cx	FFFFh to avoid default prompt

When this interrupt is issued, the drive indicated by DL is already the active unit for the shared physical device, as far as IO.SYS is concerned. IO.SYS is about to ask the user to insert the disk intended for the newly active unit in case this differs from the disk already in the physical drive. If the interrupt returns with cx = FFFFh, then some program has presumably intercepted the broadcast and issued its own prompt (and therefore, IO.SYS does not duplicate the effort).

As a side-effect of having IO.SYS skip its default prompt (and not wait for the user to press a key), this broadcast also provides a crude solution to the problem noted earlier, that Windows enhanced mode may hang when Windows applications allow IO.SYS to deal with the user-level consequences of changing the logical drive map.

Windows is certainly not ignorant of logical drive mapping as a subject, but neither is it well prepared to propagate the default disk swap message from the system's depths to its upper reaches. If higher layers do not allow for possible complications when accessing shared drives, then the IO.SYS prompt for a disk change simply gets lost. This would not be so bad if Windows applications weren't left waiting for lowly DOS to complete the disk access, wholly oblivious to DOS's attempts to alert its caller that some clarification is required.

The only solution available from the DOS side of this problem is to prevent IO.SYS from waiting for a keystroke that will never seem to appear (most keyboard processing having been taken over at higher levels), although this has the drawback of leaving the user without help to track changes to the logical drive map. With the TSR techniques detailed in Chapter 6, it would be trivial to write a program that hooks int 2Fh and responds to function 4A00h by returning FFFFh in register cx. Since the situation is handled correctly for all virtual machines except the System VM (in which Windows applications run), this little TSR would be loaded through WINSTART.BAT, so it appears only in the System VM.

Building a Logical Drive Map

Suppose that DOS drives are configured as follows:

DRIVE	PHYSICAL DEVICE			REMARKS
A:	floppy disk drive 1	active	shares	5¼" high-capacity drive, defaulting to 1.2MB format
B:	floppy disk drive 2		shares	3½" high-capacity drive, defaulting to 1.44MB format
C:	hard disk 1	active		primary partition
D:	hard disk 2	active		primary partition
E:	hard disk 1	active		volume in extended partition
F:	RAM disk	n/a	n/a	not an IO.SYS drive
G:	floppy disk drive 1		shares	5¼" high-capacity drive, defaulting to 360K format
H:	floppy disk drive 2	active	shares	3½" high-capacity drive, defaulting to 720K format

The table indicates the relevant flags in the IO.SYS drive tables and also attempts some explanation of how or why this configuration has come to be. Although drives C and E are both attached to the first hard disk, they do not share the same physical device (they operate independent portions of it). Drives G and H have been set up with the DRIVER.SYS utility. Suppose that the 3½" disk drive was used most recently as drive H (perhaps to format a low-density disk without thinking about switches), so that at the moment of inspection, drives B and G are inactive. The logical drive map is:

ARGUMENT	VALUE
A:	A:
B:	H:
C:	not shared
D:	not shared
E:	not shared
F:	not shared
G:	A:
H:	H:

The following source file presents functions that work with such maps, either for any one drive or for an array of all 26 possibilities.

```
/*  drvmap.c  */
```

```
#include    <standard.h>

/*  DOS-level functions to control logical drive mapping. These follow the
    usual scheme of returning a DOS error code (or zero for success)
    directly and recording other information at a specified address. The
    drive numbers accepted by these functions are 1-based. */

static WORD _dos_getlogicalmap (WORD drive, BYTE *mapptr)
{
    BYTE temp;
    _asm {
                mov     bx,drive
                mov     ax,440Eh
                int     21h
                jc      done
                mov     temp,al
    }
    *mapptr = temp;
    _asm {
                xor     ax,ax
            done:
    }
}

static WORD _dos_setlogicalmap (WORD drive)
{
    _asm {
                mov     bx,drive
                mov     ax,440Fh
                int     21h
                jc      done
                xor     ax,ax
            done:
    }
}

/*  ----------------------------------------------------------------------  */

/*  Functions one level higher (and which work with 0-based drive numbers).
    These are interested only in non-trivial mapping of single drives. */

BYTE GetLogicalDrive (WORD drive)
{
    BYTE map;
    /*   If it can be established definitely that the specified drive shares
         its physical device with other drives, return the 0-based number of
         the drive currently in control of that physical device. */

    if (_dos_getlogicalmap (drive + 1, &map) == 0 AND map != 0)
         return (map—(BYTE) 1);

    /*   In any other circumstance, return the value 0xFF. */
    return (0xFF);
}

/*  The next function tells the responsible device driver to regard the
    designated 0-based drive as the one in charge, should it be sharing its
    physical device with other drives. */

BOOL SetLogicalDrive (register WORD drive)
{
    BYTE map;

    /*   If the logical map facility is not supported for the drive, assume
         this is because it is not needed. */

    if (_dos_getlogicalmap (drive + 1, &map)) return (TRUE);

    /*   If the drive is known not to share its physical device, then again,
         the logical drive is already set. */

    if (map == 0) return (TRUE);

    /*   If the drive is already the active drive for its shared device, then
         nothing more need be done. */

    if ((WORD) map == drive + 1) return (TRUE);

    /*   Otherwise, put the new drive in charge. Failure here, given that
         the device driver supports the Get Logical Drive Map facility, is
         best regarded as an error. */

    return (_dos_setlogicalmap (drive + 1) == 0 ? TRUE : FALSE);
}

/*  -------------------------------------------------------------------------  */

/*  The following two functions work with the complete drive map. */
```

```
void SaveDriveMap (register BYTE *drivemap)
{
    register WORD drive;
    for (drive = 0; drive < 26; drive ++) {
          *drivemap ++ = GetLogicalDrive (drive);
    }
}

void RestoreDriveMap (register BYTE *drivemap)
{
    register WORD drive;
    for (drive = 0; drive < 26; drive ++) {
          if ((WORD) *drivemap ++ == drive) SetLogicalDrive (drive);
    }
}
```

Inactive Shared Drives and GetDriveType in Windows

Windows programmers have access to the logical drive map through the documented `GetDriveType` API function. This takes a 0-based drive number as its argument and is documented as returning the word 2, 3 or 4 to describe the drive as removable, fixed or remote, respectively. The drive's value under the logical map is ascertained by SYSTEM.DRV (whose undocumented `InquireSystem` function is at the heart of `GetDriveType`), then returned in register dx, that is, as the high part of an undocumented *dword* return value for `GetDriveType`.

For shared drives that are not currently in active control of their physical device, dx contains the 1-based number of the drive that is in control. Moreover, this inactive shared-drive case alters the documented value returned by the function, so that 0 and 1 denote removable and fixed drives instead of 2 and 3. Windows applications that check only the documented word returned by `GetDriveType` will therefore treat as nonexistent any floppy drive that is currently inactive. Thus, these drives do not appear in lists of available drives such as presented in the Browse dialog box or given as icons by File Manager.

Scanning Available Drives

A prime candidate to use the whole drive map is a DRVINFO utility that examines the system's valid drives. It is assumed here that the program is not especially interested in a drive's contents, but that interrogation may nonetheless require disk access. As each DOS drive is inspected in turn, the Get/Set Logical Drive Map functions are called (via the `_dos_` functions in the previous module) to ensure that access to the drive will not trigger an unwelcome message about disk insertion. The whole logical drive map is saved before the inspection begins and restored when the program terminates. The int 23h and int 24h exceptions are trapped to make sure that the drive map is restored before program termination. Here is the top-level module for the DRVINFO utility:

```
/* main.c — primary file for drvinfo.exe */

#include    <dos.h>
#include    <signal.h>
#include    <stdarg.h>
#include    <stdio.h>
#include    <stdlib.h>
#include    <string.h>

#include    <standard.h>

static CHAR description [] = "\n\
Presents information about DOS drives\n\
\n\
  DRVINFO [[drive:]...]\n\
";

extern BOOL IsDriveDefined (WORD);
extern void DescribeDrive (WORD, WORD);

extern void RestoreDriveMap (BYTE *);
extern void SaveDriveMap (BYTE *);
extern BOOL SetLogicalDrive (WORD);

static BYTE drivemap [26];

static void cleanup (void);
static void CtrlCTrap (int);
static int _far HardErrorTrap (void);

void PutError (CHAR *, ...);

/*  ==========================================================================  */

int main (int argc, char **argv)
{
    register CHAR *ptr;
    register WORD drive;
    BOOL listdrives;
    BYTE infowanted [26];
```

```
if (MK_WORD (_osmajor, _osminor) < MK_WORD (3, 20)) {
      PutError ("requires DOS version 3.20 or higher");
      return (-1);
}

/*    Parse the command line. The chief objective is to read a list of
      drives about which information is sought. */

memset (infowanted, FALSE, sizeof (infowanted));
listdrives = FALSE;

while (++ argv, -- argc) {
      ptr = *argv;
      if (*ptr == '/') {
          ptr ++;
          if (ptr [0] == '?' AND ptr [1] == '\0') {
                printf (description);
                return (0);
          }
          PutError ("invalid switch—%s", -- ptr);
          return (-1);
      }
      else {
          drive = (ptr [0] & ~0x20)–'A';
          if (drive <= 'Z'–'A' AND ptr [1] == ':' AND ptr [2] == '\0') {
                infowanted [drive] = TRUE;
                listdrives = TRUE;
                continue;
          }
          PutError ("invalid parameter—%s", ptr);
          return (-1);
      }
}

/*    If the command line contained no drive specifications, assume that
      information is required for all drives. */

if (NOT listdrives) memset (infowanted, TRUE, sizeof (infowanted));

/*    Take undefined drives out of the list and exit from the program if
      no valid drives remain. */

listdrives = FALSE;
for (drive = 0; drive < 26; drive ++) {
      if (infowanted [drive]) {
            infowanted [drive] = (BYTE) IsDriveDefined (drive);
            if (infowanted [drive]) listdrives = TRUE;
        }
    }
    if (NOT listdrives) {
        PutError ("no valid drives in list");
        return (-1);
    }

    /*  Save the logical drive map and make the necessary provisions to
        ensure restoration when the program terminates (or is terminated).
        A reasonable course would be to inhibit critical error generation
        and also defeat the fatal action of Ctrl-C, but it is only a small
        effort to support orderly termination via Ctrl-C. */

    #ifndef _NO_DRIVE_MAP_
    SaveDriveMap (drivemap);
    #endif

    atexit (cleanup);
    signal (SIGINT, CtrlCTrap);
    _harderr (HardErrorTrap);

    /*  For each drive for which a description has been requested, call an
        external function to get information and display it:  this division
        means that the drive-by-drive description can be changed in nature
        without affecting this module. */

    for (drive = 0; drive < 26; drive ++) {
        if (infowanted [drive]) {

            /*  It eases matters greatly that the external function be
                entitled to assume that the drive in question actually has
                logical control of its physical device. If the drive shares
                its physical device with other logical drives and does not
                happen to be the one currently in control, alter the logical
                drive map before commencing any investigation. */

            #ifndef _NO_DRIVE_MAP_
            SetLogicalDrive (drive);
            DescribeDrive (drive, drivemap [drive]);
            #else
            DescribeDrive (drive, 0xFF);
            #endif
        }
    }

    printf ("\n");
    return (0);
}

static void cleanup (void)
{
    #ifndef _NO_DRIVE_MAP_
    RestoreDriveMap (drivemap);
    #endif
}

static void CtrlCTrap (int sigint)
{
    signal (SIGINT, SIG_IGN);
    PutError ("terminated by user");
    exit (-1);
}

static int _far HardErrorTrap (void)
{
    return (_HARDERR_FAIL);
}

void PutError (CHAR *str, ...)
{
    va_list va_start (argptr, str);
    printf ("\nDRVINFO fatal error:  ");
    vprintf (str, argptr);
    printf ("\n");
    va_end (argptr);
}
```

Presenting Drive Descriptions

A fair bit of information about available drives can be obtained quickly through two DOS functions not mentioned so far. These are often the first point of call in a program that needs simply to provide a list of drives without caring too much about any finer details. In principle DOS function 4409h exists to distinguish remote drives from local ones (its documented name being "Is Drive Remote"), but the function serves equally well as a simple and noninvasive method of establishing a DOS drive's existence:

Table 16-18: DOS Function 4409h—Get Drive Flags

INPUT:

AX	4409h	
BL	1-based drive number or 00h to denote default	

SUCCESS:

cf	clear	
ax	corrupt	
dx	Flags. For local drives, bits other than those below are taken from the device driver attribute word (obviously, the bit masked by 1000h must be clear in this case). For remote drives, the other bits are all zero.	
	0200h	drive BL is shared across network via the int 2Ah interface
	1000h	drive BL is remote
	8000h	drive BL substitutes for a directory

FAILURE:

cf	set	
ax	000Fh	if drive BL is invalid

DOS 5 is the first version to ensure that the other bits are zero when the drive is remote.

In DOS 4 a drive may be remote in the sense of not using the native FAT file system but still have a DPB and device driver. If so, then the bit masked by 1000h is clear and the other bits are taken from the device driver attribute as if the drive were local. The bit masked by 0200h may spuriously describe drive Z if redirection of drive BL is paused (because an error code of 1Ah is not trapped but is treated as a drive number).

If executed under DOS 4, a number of Microsoft programs reject this function as a means to test whether or not a drive is remote (the function's documented purpose), preferring instead to inspect the CDS array directly or to look through the Assign List returned by int 21h function 5F02h. For example, SMARTDRV.EXE during its initialization looks over all the drives to decide which ones will be cached; remote drives are not cached. Installable file systems (IFS) in DOS 4 presumably prevents int 21h function 4409h from knowing everything it should.

A successful return from this function indicates only that DOS relates the given drive to some block storage that passes for "physical" at this level. It says little about how the drive fits into the overall directory structure, and, indeed, it is possible for a drive to exist in the sense of this function but not be visible to users of directory-based DOS services.

An easily constructed example (not taken up with much enthusiasm) occurs when a physical drive is JOINed to a directory: Most DOS functions will refuse direct access to the drive, requiring you instead to access it via the JOINed directory.

Another handy DOS function for obtaining salient characteristics quickly is int 21h function 4408h, which determines whether the drive in question supports removable media. It seeks no other information and is well-suited to situations in which programs need know little more about a drive. For instance, a disk caching program may use a positive result to eliminate the drive from further consideration.

Table 16-19: DOS Function 4408h—Query Media Removability

INPUT:

AX	4408h	
BL	1-based drive number or 00h to denote default	

SUCCESS:

cf	clear	
ax	0000h	drive's storage media is removable
	0001h	drive's storage media is fixed

FAILURE:

cf	set	
ax	0001h	if drive BL is remote; or if device driver does not support the Open, Close, and Removable Media commands
	000Fh	if drive BL is invalid

This function frames a device driver request packet for command 0Fh. The result returned in ax is taken from the busy bit in the device driver status word. Any device driver error is ignored. The device driver is not called at all unless bit 11 (indicating support for the Open Device, Close Device, and Removeable Media requests) is set in its attribute word.

DOS 4 only allows this function to work on remote drives as long as the installable file system (IFS) driver has bit 11 clear in its attribute word to indicate that the storage device is local. This means that the native FAT file system has been replaced, but the DOS kernel still associates the drive with a DPB and device driver. In this case, bit 11 must still be set in the device driver attribute word to indicate support for the Open, Close, and Removable Media commands.

Thus far, the growing DRVINFO program has a top end to coordinate the drive examination and modules from earlier in the chapter that will provide low-level information about individual drives, particularly the correspondence between DOS drive letter and physical drive (as understood by the BIOS disk services). One more module is needed to link with the others, to provide the routine that actually presents the information obtained for each drive.

```
/* drvinfo.c */

#include    <stdarg.h>
#include    <stdio.h>
#include    <stdlib.h>
#include    <string.h>

#include    <cds.h>
#include    <dpb.h>
#include    <driver.h>
#include    <standard.h>
#include    <sysvars.h>

/*  Some drive features are determined externally. */

extern BYTE GetBIOSDrive (WORD);
extern struct DPB _far *GetDPB (WORD);
extern void DescribeCDSProperties (WORD);

/*  For message output  */

void PutComment (CHAR *, ...);
static BOOL firstcomment;

/* ====================================================================== */

/*  C language equivalents for int 21h functions 4408h and 4409h. Like the
    DOS functions themselves, these expect 1-based drive numbers. They
    follow the standard scheme of returning a DOS error code as the function
    value and passing other information at an address supplied as the last
    argument. */

static WORD _dos_getdrivetype (WORD drive, WORD *ptr)
{
    WORD temp;
    _asm {
        mov     bx,drive
        mov     ax,4409h
        int     21h
        jc      done
        mov     temp,dx
    }
    *ptr = temp;
    _asm {
        xor     ax,ax
    done:
    }
}

static WORD _dos_removablemedia (WORD drive, WORD *ptr)
{
    WORD temp;
    _asm {
        mov     bx,drive
        mov     ax,4408h
        int     21h
        jc      done
        mov     temp,ax
    }
    *ptr = temp;
    _asm {
        xor     ax,ax
    done:
    }
}

/* ------------------------------------------------------------------ */

BOOL IsDriveDefined (WORD drive)
{
    WORD type;

    /*  If int 21h function 4409h returns an error, then the drive is
    unknown to the DOS kernel. This is the simplest possible way to
    establish a drive's current existence. */

    return (_dos_getdrivetype (drive + 1, &type) == 0 ? TRUE : FALSE);
}

/*  The next function lists drive properties. It assumes that the drive
    number given as the first argument is currently in control of its
    physical device, this having been arranged by the caller, if necessary.
    The second argument gives the drive that was originally in control of
    the physical device. */
```

```
void DescribeDrive (WORD drive, WORD activedrive)
{
    WORD biosdrive, driveflags, errorcode, temp;
    struct DPB _far *dpbptr;

    errorcode = _dos_getdrivetype (drive + 1, &driveflags);
    if (errorcode) return;

    /*  The drive exists, so list it. The first description of the
    drive will follow on the same line. Subsequent comments are written
    on new lines - this is taken care of by the PutComment function,
    which should be used for all remarks about the drive. */

    printf ("\n%c:  ", drive + 'A');
    firstcomment = TRUE;

    /*  DOS function 4409h returns flags in register dx. If the bit masked
    by 0x1000 is set, then all access to the drive is handled via the
    network redirector interface. */

    if (driveflags & 0x1000) {
        PutComment ("remote/network drive");
        return;
    }

    /*  Local drives may divided into those regarded as physical at DOS
    kernel level because they correspond directly to a unit of some
    block device driver and virtual drives that exist only as artifacts
    of the directory system (e.g., a drive created with the SUBST
    utility). Physical drives have their own DPB structure. */

    dpbptr = GetDPB (drive);
    if (dpbptr != NULL AND (WORD) dpbptr -> cDrive == drive) {
    /*  If the drive is a phantom (i.e., if it shares a physical device
        with some other drive and is not currently in control of that
        device), then just note the correspondence. */

    if (activedrive != 0xFF AND activedrive != drive)
        PutComment ("sharing device with drive %c", activedrive + 'A');
    else {
        /*  Begin by describing the physical storage device to which the
        DOS drive refers. For the first attempt, try to identify
        the drive number number used for int 13h access. */

        biosdrive = GetBIOSDrive (drive);
        if (biosdrive != 0xFF) {
            if (biosdrive >= 0x80)
                PutComment ("hard disk %d", biosdrive - 0x80 + 1);
            else PutComment ("floppy disk drive %d", biosdrive + 1);
        }
        else {
            /*  It may be that int 13h is not used to control the
                storage device. In many cases, such as the increasingly
                common CD-ROM drives, documented interfaces exist
                through which this program could obtain information if
                it were being developed further.

                Ironically, the most difficult medium to be certain of
                categorising correctly may well be the ubiquitous RAM
                disk:  no interface is provided, but a reasonable test
                (employed by several Microsoft products) seems to be
                that only with a safe medium like RAM would a drive with
                an FAT system risk having only one FAT copy. */

            if (_dos_removablemedia (drive + 1, &temp) != 0) temp = -1;

            if (temp != 0 AND dpbptr -> cFATs == 1)
                PutComment ("RAM disk");
            else
                PutComment ("unknown %s drive",
                    temp == 0 ? "removable" : "fixed");
            }
        }
    }
    DescribeCDSProperties (drive);
}

void PutComment (CHAR *str, ...)
{
    va_list va_start (argptr, str);
    if (NOT firstcomment) printf ("\n    ");
    else firstcomment = FALSE;
    vprintf (str, argptr);
    va_end (argptr);
}
```

Little more would be required for the program's makefile than a list of the modules involved, were it not for the allowance made for conditional compilation. Several component modules offer a choice of methods:

```
!INCLUDE    <standard.mak>

TARGETS : SYMBOLS drvinfo.exe

#   Define the following symbols as null macros on the NMAKE command line:
#
#   NOIOSYS         to skip undocumented reference to IO.SYS drive tables and
#                   force use of int 13h trap;
#
#   DOSAPI          to use int 21h functions rather than inspect internal
#                   structures;
#
#   NODRVMAP        to see the effect of ignoring logical drive mapping.
#
#   The choice of symbols used for a make is remembered so that object
#   modules can be deleted selectively next time the executable is rebuilt.
#   The information is prepared in the from of macro definitions and written
#   to a file called SYMBOLS.LST for use as a response file to NMAKE. The
#   project should therefore be built with a command line like
#
#         NMAKE /f drvinfo.mak NOIOSYS= NODRVMAP= @symbols.lst
#
#   If the macros in SYMBOLS.LST have not been loaded, then all object
#   modules are deleted to force complete recompilation.

OLD = OLD_SYMBOLS=
!IFDEF NOIOSYS
OLD = $(OLD) NOIOSYS_OLD=
!ENDIF
!IFDEF DOSAPI
OLD = $(OLD) DOSAPI_OLD=
!ENDIF
!IFDEF NODRVMAP
OLD = $(OLD) NODRVMAP_OLD=
!ENDIF

SYMBOLS :
  @rem >nul <<symbols.lst
$(OLD)
<<KEEP
!IF defined(OLD_SYMBOLS)
!IF defined(NOIOSYS) ^^ defined(NOIOSYS_OLD)
  del biosdrv.obj
!ENDIF
!IF defined(DOSAPI) ^^ defined(DOSAPI_OLD)
```

```
  del doslevel.obj
!ENDIF
!IF defined(NODRVMAP) ^^ defined(NODRVMAP_OLD)
  del main.obj
!ENDIF
!ELSE
  del *.obj
!ENDIF

biosdrv.obj : biosdrv.c
!IFDEF NOIOSYS
  $(COMPILE) /D _NO_IOSYS_ biosdrv.c
!ENDIF

break.obj : break.c

doslevel.obj : doslevel.c
!IFDEF DOSAPI
  $(COMPILE) /D _USE_DOS_API_ doslevel.c
!ENDIF

!IFNDEF NODRVMAP
drvmap.obj : drvmap.c
!ENDIF

drvinfo.obj : drvinfo.c

main.obj : main.c
!IFDEF NODRVMAP
  $(COMPILE) /D _NO_DRIVE_MAP_ main.c
!ENDIF

OBJECTS = biosdrv.obj break.obj doslevel.obj drvinfo.obj main.obj
!IFNDEF NODRVMAP
OBJECTS = $(OBJECTS) drvmap.obj
!ENDIF

drvinfo.exe : $(OBJECTS)
  $(LINK) $(OBJECTS),drvinfo;
```

A possibly atypical result of running the finished program is

```
P:\DRIVES\DRVINFO>drvinfo

A:  floppy disk drive 1
```

```
B:  floppy disk drive 2
    sharing device with drive H
C:  hard disk 1
D:  hard disk 2
E:  hard disk 1
F:  RAM disk
G:  floppy disk drive 1
    sharing device with drive A
H:  floppy disk drive 2
I:  substituted for D:\C600\INCLUDE
L:  substituted for D:\C600\LIB
```

where in fact, the drive substitutions continue at some length but have been truncated. If nothing else, the list of drives shows that the author is quite happy to configure logical floppy drives for testing or demonstration but can't imagine a plausible reason for using the JOIN utility. More seriously, readers interested in assessing the technical quality of software with respect to disk handling should also configure a logical floppy, as just one of those slightly unusual circumstances that tend to be missed by developers and testers alike. To see the effect of ignoring Logical Drive Mapping, rebuild DRVINFO with the NODRVMAP and NOIOSYS symbols (as shown in the makefile).

DRVINFO lays no claim to comprehensiveness. For instance, it does not deal with utilities such as ASSIGN (DRVINFO would need to consult ASSIGN's int 2Fh function 0601h to get the drive-assignment table), nor as a more important example does it know anything *special* about NetWare drives (i.e., beyond what Novell's int 21h handlers return as though the information were coming from the DOS kernel). DRVINFO is presented only to solidify those parts of this chapter which would otherwise be only background for the general purpose of writing a floppy disk utility in Chapter 18. Readers are invited to expand DRVINFO so that it reports any other drive characteristics they find interesting. Particularly for physical drive properties, this should be no trouble using the Generic IOCTL interface examined in the next section.

Generic IOCTL

As has been noted several times already in this chapter, it is generally preferable to use the DOS block-device generic IOCTL services rather than use int 25h, int 26h, or int 13h. These services made their first appearance in DOS 3.2 and are used by such DOS utilities as FORMAT and CHKDSK.

Table 16-20: DOS Function 440Dh—Generic IOCTL for Block Device Drivers

INPUT:

AX	440Dh
BL	1-based drive number or 00h for current drive
CH	08h

Table 16-20: Continued

CL	minor code (see below)
DS:DX	address of transfer buffer for control data

SUCCESS

cf	clear
ax	corrupt

FAILURE

cf	set	
ax	0001h	if CH ≠ 08h; or if drive BL is remote; or if device driver does not support Generic IOCTL
	0005h	if device driver returns an error
	000Fh	if drive BL is invalid

The function frames a device driver request packet for command 13h (Generic IOCTL, naturally). The device driver attribute word must have bit 6 set to indicate support for Generic IOCTL (the same bit required for Logical Drive Mapping). DOS does not call the driver if this bit is not set; this is also true if CH ≠ 08h.

DOS 4 only allows generic IOCTL for remote drives as long as the installable file system (IFS) driver has bit 11 clear in its attribute word to indicate that the storage device is local. This means that the native FAT file system has been replaced but the DOS kernel still associates the drive with a DPB and device driver. In this case (also for the conventional IOCTL interface and the parallel services for character devices), the Generic IOCTL command is not passed to the device driver (as for the Logical Drive Map functions) but to the IFS driver using int 2Fh function 112Bh.

A growing suite of commands (referred to as minor codes) is offered so that DOS programs can access block storage beneath the file system but still retain some measure of hardware independence. The commands currently available from the default block device driver in IO.SYS are

MINOR CODE	PURPOSE
40h	Set Device Parameters
41h	Write Track
42h	Format Track
46h	Set Media ID

47h	Set Access Flag
60h	Get Device Parameters
61h	Read Track
62h	Verify Track
66h	Get Media ID
67h	Get Access Flag
68h	Sense Media Type

Querying Support for Specific Minor Codes

Starting with DOS 5, programs may be able to use int 21h function 4411h to ask the device driver responsible for a given drive whether or not it supports a particular Generic IOCTL command. A matching interface (function 4410h) is available for character device drivers.

Table 16-21: DOS Function 4411h—Query Block Device Driver's IOCTL Support

INPUT:

AX	4411h
BL	1-based drive number or 00h for current drive
CH	08h
CL	minor code

SUCCESS:

cf	clear
ax	corrupt

FAILURE:

cf	set	
ax	0001h	if CH ≠ 08h; or if drive BL is remote; or if device driver does not support both the Generic IOCTL and IOCTL Query commands
	0005h	if device driver returns an error—rejection of the minor code is indicated by the Invalid Device Request error
	000Fh	if drive BL is invalid

Device drivers are not required to support this query facility. For both character and block device drivers, bit 7 of the attribute word must be set to signify this support, which of course, is pointless if bit 6 does not also indicate that a Generic IOCTL interface exists. As the table shows, DOS returns its Invalid Function error if these bits are not both set: the device driver is not called. The query should therefore be understood only in a negative sense: Does the device driver specifically reject this minor code? As shown in `_dos_gioctl` in GIOCTL.C below, applications should inspect the error code before assuming that the Generic IOCTL command in question is not supported.

An example distributed as standard with DOS is DRIVER.SYS, discussed earlier because of its intimate relationship with the default block device driver in IO.SYS. Essentially, DRIVER.SYS is a stub which provides a Drive Table and a small amount of code that passes all device driver calls to the default driver. However, only bit 6 is set in the DRIVER.SYS attribute word, so an IOCTL Query about a drive defined by DRIVER.SYS always fails, even though a properly framed Generic IOCTL call is likely to succeed (being handled by IO.SYS).

The following module provides a C interface to some of the block-device generic IOCTL services:

```
/* gioctl.c */

#include    <stdlib.h>

#include    <doserror.h>
#include    <drivparm.h>
#include    <standard.h>

/*  The following functions invoke the Generic IOCTL interface with block
    device drivers. They return zero on success and indicate failure
    through a normal DOS error code. For greater definition, extended error
    information should be sought immediately after the function returns.
    Device driver errors occurring during Generic IOCTL functions do not
    generate int 24h exceptions. */

static WORD _dos_gioctl (BYTE minorcode, WORD drive, void _far *data)
{
    _asm {
            mov     ch,08h      ; special value for disk services
            mov     cl,minorcode
            mov     bx,drive

    /*  If the DOS version is at least 5, check first that the device driver
        supports this particular minor code. */

            cmp     _osmajor,5
            jb       ok

            mov     ax,4411h
            int     21h
            jnc     ok

            cmp     ax,E_ACCESS_DENIED
            jz       done
        ok:

    /*  If the minor code is not specifically rejected, proceed with the
        Generic IOCTL command. */

            push    ds
            lds     dx,data
            mov     ax,440Dh
            int     21h
            pop     ds
            jc        done
            xor     ax,ax
        done:
    }
}

WORD _dos_getdrivparm (WORD drive, struct DRIVPARM *parmptr)
{
    return (_dos_gioctl (0x60, drive, parmptr));
}

WORD _dos_setdrivparm (WORD drive, struct DRIVPARM *parmptr)
{
    return (_dos_gioctl (0x40, drive, parmptr));
}

WORD _dos_readtrack (WORD drive, struct RWBLOCK *rwptr)
{
    return (_dos_gioctl (0x61, drive, rwptr));
}

WORD _dos_writetrack (WORD drive, struct RWBLOCK *rwptr)
{
    return (_dos_gioctl (0x41, drive, rwptr));
}

WORD _dos_getmediaid (WORD drive, struct MEDIA_ID *idptr)
```

```
{
    return (_dos_gioctl (0x66, drive, idptr));
}

WORD _dos_setmediaid (WORD drive, struct MEDIA_ID *idptr)
{
    return ( dos_gioctl (0x46, drive, idptr));
}
```

The remainder of this chapter provides a detailed analysis of each of the eleven currently defined block-device generic IOCTL minor codes.

Device Parameters

Get Device Parameters (minor code 60h) extracts data from the IO.SYS table for the drive in question. Its alter ego (Set Device Parameters, minor code 40h) enters data selectively in the same table. In both cases bit flags in the first byte of the parameter block passed to the driver determine the nature of information sent or received in the remaining fields. Refer to the structure given earlier in the chapter for the IO.SYS Drive Tables ("The Default Block Device Driver") for more details about the drive data being accessed through these Generic IOCTL commands.

Table 16-22: Data Structure Returned by Get Device Parameters

OFFSET	SIZE	DESCRIPTION
00h	byte	If bit 0 is clear, then the BPB returned at offset 07h is the drive default. If bit 0 is set, then the BPB reflects the drive's current contents—for removable media, this usually involves an attempt to read the boot sector and possibly the disk second sector (in the hope of identifying the disk from the FAT media descriptor) An inability to determine the drive's current contents is indicated by returning an appropriate device error.
01h	byte	Device type—also known as the form factor admitted by the **drivparm** statement in CONFIG.SYS and on the command line for DRIVER.SYS. This is a property of the drive, not its current contents.
02h	word	The low 2 bits of a more informative status word maintained by the default block device driver. Bit 0 is set for a fixed drive and bit 1 is set if the BIOS supports a floppy drive's change-line capability.
04h	word	The maximum number of cylinders the drive supports. This is a property of the drive not its current contents.
06h	byte	The default block device driver always returns zero in this field.
07h	19h	A BIOS Parameter Block—either for the current contents of the drive or for the format that the block device driver regards as the default for the drive, depending bit 0 in the control byte.

The BPB returned by this command when bit 0 of the control byte is clear is an exact representation of the disk as understood by IO.SYS, in the sense that it is copied directly from the internal table for the drive. Recall that this need not be an exact copy of the BPB on disk.

This extra material concerned with track layout is *always* processed, whatever the state of bit 1 in the control byte. The information, both the number of sectors and the array of sector descriptors that comprises the track layout, is saved as global data in IO.SYS and affects the operation of subsequent track-level commands that read, write, format, or verify, whatever the drive.

Table 16-23: Data Structure Expected for Set Device Parameters

OFFSET	SIZE	DESCRIPTION
00h	byte	If bit 0 is set, then subject to the effect of bit 1 described below, the BPB at offset 07h will be adopted as the default format for the drive. A clear bit 0 directs the default block device driver to remember the BPB at offset 07h as the disk characteristics and not to attempt discovery of the actual contents of the drive in future—this directive is cancelled by setting the default BPB for the drive. If bit 1 is set, the function deals only with track layout, ignoring the BPB and ind all fields before offset 26h. Set bit 2 to indicate that sectors are equally-sized (good track layout). Without this knowledge, set explicitly for the drive, the default block device driver processes Read/Write Track requests sector by sector, incurring a significant time penalty if in fact the sectors are all the same size.
01h	byte	Device type (form factor).
02h	word	Only the low 2 bits mean anything to the function. An attempt to set change-line support (bit 1) will be ignored unless at least one drive supported it at startup.
04h	word	The maximum number of cylinders the drive supports.
06h	byte	This value affects the subsequent behavior of Format Track when dealing with a 1.2MB drive. It is used to deduce the correct DASD type if the format operation seems to lack BIOS support. Set a non-zero value to anticipate formatting a 360K disk in a high-capacity drive.
07h	1Fh	A BIOS Parameter Block—to serve as the BPB for the drive's contents (in which c only 19h bytes are used) or to be adopted as the default format for the drive, depending on bit 0 in the control byte.
26h	word	Number of sectors in the track layout that follows at offset 28h. General Failure w be returned as a device error if this number is 64 or greater. Each sector is describ by two words: a 1-based sector number, followed by the sector size in bytes (128, 256, 512, and 1024 are the expected values). The default track layout provides fo 36 sectors in the natural order, all with the usual size of 512 bytes.

Even if it were reasonable to assume of the operating environment that sector ordering and sector size are never changed from their default values, the word at offset 26h in the Device Parameters structure must still be set to zero in order to preserve the existing track layout in the IO.SYS data area (but this practice has unpleasant consequences for Format Track, which takes this to be the number of sectors per track in preference to a BPB field).

For safety, the track-level commands (following) should not be issued without first calling Set Device Parameters to force the adoption of a suitable track layout, even if no attempt is ever made to alter sector ordering or size. Observe that no interface exists by which the track layout may be inspected, for instance to determine whether or not it has changed during a task switch. Note also that no mention of this interconnection appears in the official *Programmer's Reference*. Further, a programmer who has followed the *Programmer's Reference*, and thus believes the Device Parameters structure to end with the BPB, may find the function failing mysteriously, depending on the word at offset 26h.

Disk Access Track by Track

Four services are provided for track-by-track access: write (minor code 41h), format (42h), read (61h), and verify (62h).

Table 16-24: Data Structure Expected for Read/Write Track

OFFSET	SIZE	DESCRIPTION
00h	byte	unused
01h	word	Head (only the low byte is significant)
03h	word	Track
05h	word	Starting sector
07h	word	Number of sectors to read or write
09h	dword	Address of transfer buffer

The head, track, and starting sector fields are all 0-based. The field at offset 05h is not interpreted directly as the starting sector but is used instead as an index into the track layout recorded during the last call to Set Device Parameters. The lookup is not preceded by a bounds check against the number of sectors in the track layout (remembered from the last call to Set Device Parameters).

For each drive, IO.SYS keeps a flag indicating whether or not to optimize track-level reading, writing, and verifying by assuming Good Track Layout such that all sectors on the track have equal size and that the track layout lists sectors in ascending order. The flag is controlled only via Set Device Parameters. The *Programmer's Reference* directs that the flag always be set. If any of the track-level commands to read, write, or verify are called for a drive that has not had Good Track Layout set explicitly (and note that the flag is clear for each drive by default), a separate BIOS call is made for each of the requested sectors. This decomposition allows IO.SYS to adjust the Diskette Parameter Table between sectors to accommodate variations in size such as might be found on copy-protected or non-PC diskettes. The obvious (and quite serious) time penalty if the sectors are in fact equally-sized will be encountered if Set Device Parameters is not called to set Good Track Layout before using Read/Write/Verify Track—even if the default track layout is known to be still in place. In Chapter 18, readers can experiment with this effect by running the DISKDUP program with its **/SetDrivparmOff** switch.

Apart from the indirection used to obtain the starting sector, no reference to the track layout is needed for drives with nonremovable media if **multitrack=on** (the default setting after pass 2 through CONFIG.SYS). Otherwise the track layout is consulted again to obtain the sector size. If Good Track Layout has been set, then the size need only be fetched for the starting sector. The irony is that the track layout is surely redundant in this case. If Set Device Parameters must be called, then presumably the caller will pass a BPB containing the intended bytes-per-sector value.

In the absence of Good Track Layout the present implementation is unprepared for a request to transfer zero sectors (the likely result is a system crash) and if the drive contains non-removable media, an oversight limits the read/write/verify to just one sector if **multitrack=on**.

Table 16-25: Data Structure Expected for Format Track

OFFSET	SIZE	DESCRIPTION
00h	byte	Set bit 0 to prepare the BIOS for the format and obtain a format status, without actually formatting the track. The driver supplies int 13h function 18h with the number of cylinders it believes the drive supports and the intended number of sectors per track (taken from the disk BPB). The format status is returned in this same byte: 00h if BIOS support exists and the format is acceptable 01h in the absence of BIOS support (for any reason other than a timeout) 02h if BIOS support exists but the proposed format is rejected 03h to show a timeout error If bit 0 is clear, then setting bit 1 forces the device error General Failure. If both bits 0 and 1 are clear (the official documentation advises that this byte must be zero), then an attempt is made to format the track designated by the remaining fields in this structure. Failure is indicated by an appropriate device error.
01h	word	Head (only the low byte is significant)
03h	word	Track

The default block device driver does not allow this command to format hard drives (form factor 5). If bit 0 is clear in the control byte, the command is converted to Verify Track.

Table 16-26: Data Structure Expected for Verify Track

OFFSET	SIZE	DESCRIPTION
00h	byte	Set bit 1 to verify multiple tracks with the one call—the field at offset 05h is irrelevant otherwise
01h	word	Head
03h	word	Track
05h	word	Number of tracks to verify (only if bit 1 is set in the control byte at offset 00h). The equivalent number of sectors must be less than 256, or else the function returns General Failure.

Media Identification

Starting with DOS 4 it is possible to maintain a volume label independently of the file system. In earlier versions the volume label is a special entry in the disk's root directory—marked with the file attribute 08h, possibly combined with others such as read-only. For standard storage media operated by the default block device driver in IO.SYS, the new-style volume label, a 32-bit serial number, and an eight-character name for the disk's file system are all placed in the boot sector as an extension to the BPB as indicated in the earlier section on "The Boot Sector" and summarized here for convenience:

Table 16-27: Components of Media ID in Boot Sector

OFFSET	SIZE	DESCRIPTION
0Bh	19h	a BIOS Parameter Block (BPB)
26h	byte	the signature value 29h indicates validity of the media identifiers in the remaining fields
27h	dword	serial number
2Bh	0Bh	volume label
36h	08h	file system description

Generic IOCTL minor codes 46h and 66h get and set the media ID; there is also an equivalent undocumented int 21h function 69h. By providing a software interface for access to these media identifiers, DOS recognizes a need for flexibility when dealing with different forms of storage. Indeed, the media ID need not actually be on disk, let alone in the first sector. It is conceivable for instance that a device driver could retrieve a manufacturer's hard-wired serial number.

Having said that, it should probably also be observed that the DISKCOPY program distributed as standard with DOS reads and writes the serial number directly on disk as an extension to the BPB in preference to using the Generic IOCTL interface (or the equivalent DOS function). Moreover, in addition to the magic number 29h in the signature byte, it accepts the value 28h as indicating the presence of valid media identifiers in the boot sector.

Table 16-28: Data Structure Expected for Get/Set Media ID

OFFSET	SIZE	DESCRIPTION
00h	word	unused
02h	dword	Serial number
06h	0Bh	Volume label
11h	08h	File system

The default block device driver's behavior is to take the combination of serial number, volume label, and file system directly from the boot sector at offset 27h or to copy the information there, depending on the minor code. In both cases, it requires the boot sector to already have the magic number 29h as the signature byte at offset 26h, and the media descriptor at offset 15h must be at least F0h. Should these constraints not be met (e.g., a floppy disk formatted under DOS 3), then the default block device driver returns Invalid Media Type. Failure to read the boot sector (and write it, if setting the media ID) is reflected by an appropriate device error. At this point, a program would typically manipulate the old-style volume-label entry in the root directory, using int 21h function 11h (FCB Find First).

For completeness, note that the media ID may be manipulated via a DOS function of its own, equivalent to accessing it through the Generic IOCTL interface. Several programs included with DOS, such as SYS.COM, get the media ID using the undocumented int 21h function 69h call rather than the equivalent documented generic IOCTL services.

Table 16-29: DOS Function 69h—Get/Set Media ID

INPUT:

AX	6900h or 6901h for Get/Set Media ID respectively
BL	1-based drive number or 00h for current drive
DS:DX	address of transfer buffer containing Media ID structure

Table 16-29: Continued

SUCCESS:

cf	clear
ax	corrupt

FAILURE:

cf	set	
ax	0001h	if drive BL is remote; or if device driver does not support Generic IOCTL
	0005h	if device driver returns an error
	000Fh	if drive BL is invalid

DOS 4 only allows generic IOCTL for remote drives as long as the installable file system (IFS) driver has bit 11 clear in its attribute word to indicate device driver support.

Get/Set Access Flag

Microsoft's DOS 6 *Programmer's Reference* refers without much explanation to a SPECIALFUNC structure used by generic IOCTL minor codes 67h and 47h (Get/Set Access Flag). As shown below, this structure contains nothing more than the drive access flag. As noted earlier, the FORMAT utility uses minor code 67h to enable the drive access flag after it has successfully initialized the structures required to support a DOS file system. Even when the access flag is disabled, FORMAT can still perform track-by-track access.

Table 16-30: Data Structure Expected for Get/Set Access Flag

OFFSET	SIZE	DESCRIPTION
00h	byte	unused
01h	byte	The access flag is passed in this byte as 0 or 1. If zero, all attempts to call the normal device driver commands Read, Write, Write with Verify or the Generic IOCTL commands Get/Set Media ID produce the device error Invalid Media Type—note however, that track-level access is not prohibited. Setting a nonzero value permits drive access as normal.

Media Type

Finally, DOS 5 introduced a new IOCTL minor code (68h) that has considerably less relevance to most machines than the description in the official literature might suggest. The default block device driver consults an int 13h function 20h that is unlikely to be defined nontrivially for many BIOSes at the time of writing.

Table 16-31: Data Structure Expected for Sense Media Type

OFFSET	SIZE	DESCRIPTION
00h	byte	The value 01h is returned in this byte if int 13h function 20h determines that the drive contains its default media type. Otherwise, the value returned is zero.
01h	byte	If possible, the form factor for the drive is returned here. Otherwise, the value returned is zero. The default block device driver is presently able to return form factors 2, 7 and 9 (for increasing densities of 3.5" disks). Possible device errors are Not Ready and Invalid Media Type.

Int 13h function 20h is expected to identify drive types according to the usual CMOS scheme. In particular, the function returns with al from the following table:

CMOS DRIVE TYPE	COMMON DESCRIPTION	FORM FACTOR ASSIGNED
1	360K	
2	1.2MB	
3	720K	2
4	1.44MB	7
6	2.88MB	9

The int 13h function is expected to use error code ah = 32h to denote that a valid drive type is in al but that the drive does not contain its default media type.

When available (for example, when implementations of int 13h function 20h become more common), the Sense Media function should be a fool-proof way of detecting whether the disk drive is empty or contains an unformatted disk.

CHAPTER 17

Error Codes

The DISKDUP program we are building in this final part of the book will need to deal with a variety of errors. A program's reaction to operating system errors often reveals much about its authorship. It is undeniably easier to get a program to "do" more if less time is devoted to rigorous error checking. Of course, a program written with this attitude may fail if executed in an environment just a little different from the developer's. Even programs from professional software houses sometimes betray a development cycle in which sound error handling has hardly figured.

This extreme case in which possible error conditions are ignored need not be considered at length here. If anything, it is more likely that readers, being too keenly aware of the potentially disastrous result of leaving an error unchecked, develop their work with a worst-case attitude in which all system calls are inspected for error conditions, all forms of which are treated according to the worst possible outcome. The result of course, is hard and unforgiving, but solid error checking is an essential feature of quality programming and must be built-in from the start. Later, as the product nears completion, thought can be given to differentiating error conditions into the genuinely fatal and those which offer a chance of recovery (either by the program internally or by user intervention).

Two concerns are addressed here. One is that the program be able to resolve the nature of any system error. It is vital both that the system return enough information at run-time and that the accompanying documentation be sufficiently accurate to support the programmer's well-intentioned endeavor. This first theme will be taken up shortly, but a more pressing concern for an author trying to discuss error codes is the question of description.

Error Code Translation

Inevitably programmers must confront the possible occurrence of low-level errors. To some extent, problems that occur inside a DOS call (critical errors) can be left to the critical error handler provided by COMMAND.COM. This is a default facility quite unsuited to programs that prepare their own displays. Increasingly software is expected to replace the default handler so that errors are presented in the most convenient way. For this type of error it would usually suffice to reproduce the default handler's description in a pop-up window or at some preset location on the screen—perhaps without the infamous "Abort, Retry, Ignore, Fail?," but with the same error message.

a pop-up window or at some preset location on the screen—perhaps without the infamous "Abort, Retry, Ignore, Fail?," but with the same error message.

At the other end of the spectrum are error conditions that reflect a failure to obtain a system resource and are not strictly the program's own problem but can nonetheless be accommodated by the machinery used for the program's high-level errors. For instance, it is entirely the application's own problem if an important file is opened and read successfully but is found to contain data in an unexpected format. The application *must* generate its own description. Inability to open the file in the first place may be a system error, but the application could treat the problem as one of its own making. This has the advantage of sheltering users from low-level details; an application that is mindful of novice users may go to some lengths to offer advice such as increasing the **files** entry in CONFIG.SYS. Advice can rarely be comprehensive, however, and with the growing sophistication of users in general it is surely beneficial to include standard phrases such as "Too many open files," if only to guide knowledgeable users attempting indirect inferences.

Standard Error Messages

The natural place for a program to look for a set of standard error descriptions is COMMAND.COM itself—for the messages used by its criticial error handler by many of its internal commands and indeed, by many of the utilities distributed as part of DOS.

Table 17-1: Standard Messages for DOS Error Codes (as Implemented by COMMAND.COM)

ERROR	DESCRIPTION
0001h	Invalid function
0002h	File not found
0003h	Path not found
0004h	Too many open files
0005h	Access denied
0006h	Invalid handle
0007h	Memory control blocks destroyed
0008h	Insufficient memory
0009h	Invalid memory block address
000Ah	Invalid environment
000Bh	Invalid format
000Ch	Invalid function parameter

Table 17-1: Continued

ERROR	DESCRIPTION
000Dh	Invalid data
000Fh	Invalid drive specification
0010h	Attempt to remove current directory
0011h	Not same device
0012h	No more files
0013h	Write protect error
0014h	Invalid unit
0015h	Not ready
0016h	Invalid device request
0017h	Data error
0018h	Invalid device request parameters
0019h	Seek error
001Ah	Invalid media type
001Bh	Sector not found
001Ch	Printer out of paper error
001Dh	Write fault error
001Eh	Read fault error
001Fh	General Failure
0020h	Sharing violation
0021h	Lock violation
0022h	Invalid disk change
0023h	FCB unavailable
0024h	System resource exhausted
0025h	Code page mismatch
0026h	Out of input
0027h	Insufficient disk space
0050h	File exists
0052h	Cannot make directory entry

Table 17-1: Continued

ERROR	DESCRIPTION
0053h	Fail on INT 24
0054h	Too many redirections
0055h	Duplicate redirection
0056h	Invalid password
0057h	Invalid parameter
0058h	Network data fault
0059h	Function not supported by network
005Ah	Required system component not installed

Examination of this table raises several questions. Note first that this table of COMMAND.COM error messages is missing the many numbers between 27h and 50h, which correspond to network errors. Despite this, COMMAND.COM is able to find a description like "NET804: Network busy" in preference to a generic "Extended error 0036h." Clearly some means exists to obtain error strings for codes that lie outside the command processor's usual domain. We will examine this (see Table 17-3).

Second, observe that some of the errors given in the table acquired definition comparatively recently. The errors numbered 0025h, 0026h, 0027h, 0059h, and 005Ah were introduced for DOS 4. Of course the possible definition of new errors in a future DOS version is no problem for COMMAND.COM, but it seems a little unfair that an application trying its best to present errors in some sort of standard form approximating a human language should be reduced to describing new errors by number.

A related question is whether all these errors can occur as the normal consequence of a DOS call.

There are more than a few entries in the table, and it would obviously help a lot to have the error messages in COMMAND.COM also be available communally instead of having to assemble them all into each program.

Error Code Translation by COMMAND.COM

This facility for converting numbered error codes to more readily meaningful strings kept in memory by COMMAND.COM was first developed for DOS 4. In that version the interface consists of several components coordinated through the DOS kernel. The historical details are given later, but for most purposes, at least in a book concentrating on DOS 5 and 6, it suffices to know that the interface has now been trimmed as far as backward compatibility allows and moved to top-level instance of COMMAND.COM where it belongs:

Table 17-2: Int 2Fh Support for Error Translation by COMMAND.COM in DOS 5

INPUT:

AX	122Eh	
DX	0000h	for table of DOS errors
	0002h	for table of parse errors
	0004h	for table of critical errors
	0008h	for translation routine (calling protocol given in Table 17-3)

OUTPUT:

es:di	address of required object (es = 0001h indicates COMMAND.COM table)

For the table addresses, the segment component is 0001h. The only true address returned by the interrupt is the calling point for the translation. Since COMMAND.COM provides the tables and the translation routine, the segment component is of course redundant for the table addresses, which are best regarded as 32-bit handles (see ERRORMSG.C below).

The interface is intended for the private use of DOS utilities, so robustness to bad parameter passing is not a design consideration. The value passed in DX is not checked against any limit. In DOS 5 loading DX with an odd value causes the interrupt to return with nothing changed. Using even values of DX other than those given in the table above corrupts es:di.

In DOS 5 the main table of DOS errors serves as the critical error table also. In DOS 4 the two tables are disjoint.

In DOS 4 only, table addresses are kept as global property in the DOS kernel and can be entered as well as retrieved (using odd values for DX instead of even). Moreover, a fourth table (accessible using DX = 0006h) is supported for network errors, with default strings set by IFSFUNC. The ability to enter table addresses in DOS 4, combined with the absence of parameter checking, means that passing DX > 0009h corrupts kernel data.

As a historical note, some explanation should be given for this interface's design. Almost all the standard utilities distributed with DOS 4 contain library code that provides common routines for message output. This library code consults message tables in an order of precedence for each of five classes. In any one class other than the one devoted to the program's own high-level output, some tables may be provided by the program itself and some obtained through int 2Fh function 122Eh. Either way, the tables are assumed to have the same format: The library routines extract the message corresponding to a given code irrespective of the table's source. The interface is designed not so much for error translation, but merely to store and retrieve default error messages. One exception is provided: If a table's segment address is 0001h, then the decoding is passed to the resident translator, which is assumed to have entered the table address in the first place and therefore knows the real segment address. COMMAND.COM—which provides the translator—installs the message tables for the normal DOS errors, parse errors and critical errors, using 0001h for the segment component in each address. IFSFUNC.EXE—which does not provide the translator—installs the message table for network errors, but must set the full segment:offset address, since each program that consults this table is expected to perform its own decoding.

In DOS 5, the facility for storing and retrieving table addresses has been stripped from the interface, but compatibility with these library routines for message output demands that the tables provided by COMMAND.COM still be "addressed" with segment 0001h.

Observe that a program intending to use this interface to obtain standard error messages has no need to address the string tables directly, nor to know the format in which the translator stores and indexes them. Indeed, COMMAND.COM goes one step further: Unless the top-level command processor is started with its **/msg** switch, most of the error messages won't actually be in memory.

Table 17-3 presents the calling protocol for the rotuine whose address is returned by int 2Fh function 122Eh when DX = 08h.

Table 17-3: COMMAND.COM Error Translation Routine

INPUT:

AX	error code
ES:DI	address of relevant table (ES = 0001h indicates COMMAND.COM table)

SUCCESS:

cf	clear
es:di	address of error message

FAILURE:

cf	set
es:di	corrupt

The error message extracted by the translator is a Pascal-style string in which the first byte counts the characters.

The segment component passed in ES is in fact irrelevant, but has been included in the description to balance the table addresses returned by the int 2Fh part of the interface.

COMMAND.COM always retains messages for errors 0013h to 0027h inclusive. Other messages may have to be fetched from the COMMAND.COM file on disk, using the pathname designated by the COMSPEC environment variable. COMMAND.COM does not examine the environment each time it needs to reload some or all of its disk image but refers instead to an internal cache of this variable's setting (which third-party environment editors have no reliable way to update).

This error translation service is not documented officially, but with that caveat, it may appeal to readers as a simple mechanism which saves labor and standardizes the treatment of DOS error codes.

```
/* errormsg.c */

/*  This file demonstrates the retrieval of standard error messages from the
    primary instance of COMMAND.COM in DOS 5. */

#include    <stdio.h>
#include    <stdlib.h>
#include    <string.h>

#include    <standard.h>

/*  Typedef for a counted string—the form in which Pascal strings are
    stored and in which the COMMAND.COM error conversion routines provide
    their results. */

typedef struct {
    BYTE length;
    CHAR string [];
} LSTRING;

static void _far *QueryTranslator (WORD);
static LSTRING _far *GetMessage (WORD, void _far *);

static void _far *lpTranslator;
static BOOL TranslatorSought = FALSE, TranslatorFound = FALSE;

static void _far *lpDOSErrorTable;

#define DOS_ERRORS      0
#define PARSE_ERRORS     2
#define CRITICAL_ERRORS 4
#define TRANSLATOR      8

static CHAR errorstring [0x100];

/*  ====================================================================  */

CHAR *GetErrormsg (WORD errorcode)
{
    LSTRING _far *ptr;
    register WORD count;

    /*    The lookup procedure begins by finding the COMMAND.COM translator
          and also the table for DOS error codes (there is another for parsing
          errors). These need be located just once;  if they are not there
          first time, do not seek them again. */

    if (NOT TranslatorSought) {
          TranslatorSought = TRUE;
          if (_osmajor >= 5) {
               lpTranslator = QueryTranslator (TRANSLATOR);
               if (lpTranslator != NULL) TranslatorFound = TRUE;
               lpDOSErrorTable = QueryTranslator (DOS_ERRORS);
          }
    }

    /*    If the translator has been found, ask it for a message to match the
          error code supplied as this function's argument. The message is
          returned (by address) as an LSTRING, so convert it to ASCIIZ format
          in a temporary buffer. */

    if (TranslatorFound AND (ptr = GetMessage (errorcode, lpDOSErrorTable))) {
          _fmemcpy (errorstring, ptr -> string, count = ptr -> length);
          errorstring [count] = '\0';
    }
    else sprintf (errorstring, "DOS error %02Xh", errorcode);

    return (errorstring);
}

static void _far *QueryTranslator (WORD index)
{
    _asm {
              xor     di,di   ; int 2Fh function 122Eh might not be
              mov     es,di   ; supported and therefore not change es:di
              mov     dx,index
              mov     ax,122Eh
              int     2Fh
              mov     dx,es
              mov     ax,di
    }
}

static LSTRING _far *GetMessage (WORD errorcode, void _far *tableptr)
{
    _asm {
            mov     ax,errorcode
            les     di,tableptr
            call    lpTranslator
            mov     dx,es
            mov     ax,di
            jnc     done
            xor     ax,ax
            cwd
        done:
    }
}
```

Functions in both DISKID.C and DISKID.C rely on the `GetErrormsg` function provided by ERRORMSG.C.

Parse errors

For completeness, the COMMAND.COM messages that describe parsing errors are presented here.

ERROR	DESCRIPTION
0001h	Too many parameters
0002h	Required parameter missing
0003h	Invalid switch
0004h	Invalid keyword
0005h	<space>
0006h	Parameter value not in allowed range
0007h	Parameter value not allowed
0008h	Parameter value not allowed
0009h	Parameter format not correct
000Ah	Invalid parameter
000Bh	Invalid parameter combination

Their existence might have offered some advantage to programmers had DOS also offered a command line parsing routine for general use. With version 4, many standard DOS utilities grew substantially in size (without offering a compensating increase in functionality) precisely because they each carried the same parser, so no trickery is likely to uncover a communally accessible command line parser.

Operating system errors are presented to programmers as numbers, and it is helpful to have a conversion facility that produces standard descriptions. As long as programmers have to do their own command line parsing, they may as well construct their error messages directly, too.

Critical Errors

In principle critical errors are those that may arise during int 24h exceptions. In the absence of remote file systems such as networks or CD-ROM extensions, critical errors range from 0013h to 0027h inclusive (see Table 17-1). For the most part they are raised when the DOS kernel has called an external module such as a device driver or SHARE.EXE to perform some task, but has instead been alerted to an error. For remote systems, the principle is the same—an error has arisen that cannot be handled within the DOS kernel—but the redirector is given more responsibility for managing the situation, including the provision of its own error codes, to be discussed shortly.

A critical error handler receives control when the DOS kernel issues int 24h in the hope that whatever code is at the other end will either resolve the error condition or tell DOS how to continue. It is understood that the critical error handler will make only minimal use of DOS services, since there is already a DOS call in a suspended state of processing. The critical error handler cannot fetch error messages from disk in this situation, and for this reason, COMMAND.COM—which provides the system's default critical error handler—always retains messages for errors 0013h to 0027h. As a corollary, observe that a custom handler must take care not to use the resident error translation facility for error codes outside this range.

The last three in this range of critical errors do not arise during int 24h exceptions in DOS 5, and the last two (Out of input and Insufficient disk space) can occur in DOS 4 only after special preparation.

This may be worth a small digression here, for it is quite surprising that DOS has no documented provision of a system error to indicate that a file write has failed for lack of disk space. Strictly speaking, a request to write to a file does not *fail* for lack of disk space. DOS function 40h does not set the carry flag in this situation, but leaves the caller to compare the number of bytes actually written against the number requested. Similarly, an attempt to read past the end of a file is not regarded as an error by DOS function 3Fh. This strikes some programmers as inconvenient, if not problematic. DOS 4, which introduced the new error codes, allows software to indicate a preference for these conditions to be regarded as errors. Deliberate action is required since most programs will expect to check completion of read and write requests themselves and will not be prepared for a different system response.

Thus an open file may be marked by an appropriate bit flag in its System File Table (SFT) entry so that an attempt to read past the end of the file or a failure to allocate the required number of clusters when expanding the file triggers critical error 0026h or 0027h respectively. This SFT status flag (the bit masked by 0100h) is subject to manipulation by DOS functions 4400h and 4401h. When applied to a file, the first of these functions returns the current SFT status in dx, so a programmer need only set bit 8 and call the second function. In DOS 4 this facility to treat partial disk I/O as an error is undocumented. In DOS 5 it has been withdrawn altogether, but programmers may care nonetheless to make use of the new error codes' definitions to treat partial I/O as much like an error as possible and at the lowest level:

```
#include    <standard.h>

WORD _dos_write (int handle, void _far *buf, WORD count, WORD *numwrt)
{
    WORD temp;
    _asm {
              push    ds
              lds     dx,buf
              mov     cx,count
              mov     bx,handle
              mov     ah,40h
              int     21h
              pop     ds
              jc       fail
              mov     temp,ax
              cmp     ax,cx       // count == *numwrt?
              mov     ax,0027h     // new code for Insufficient disk space
              jnz     fail
    }
    *numwrt = temp;
    _asm {
              xor     ax,ax
         fail:
    }
}
```

Network Errors

Critical errors above 0027h indicate problems on a remote file system. They arise when a network redirector or installable file system encounters a problem with its NETBIOS interface, device drivers or whatever, just as the DOS kernel may run into difficulties with lower-level modules. Remember that these remote systems are processing int 21h functions on behalf of the DOS kernel, trying to match their operations to those which would have be conducted for a file or device with direct kernel support. When they encounter a critical error, they raise the same int 24h exception. In fact, they call the DOS kernel (int 2Fh function 1206h) to issue the exception for them, guaranteeing that it be identical to a native critical error. The only difference lies in the error information imparted to the int 24h handler. On the whole, the error codes and their interpretations will be the redirector's own constructions. This in turn presents a problem for critical error handlers: How is the error to be described at user level?

Cynics will have no trouble understanding that the problem is resolved by having redirectors offer an undocumented interface—known to COMMAND.COM and used for its default critical error handler, but beyond the normal reach of programmers trying to write custom handlers.

Table 17-4: Int 2Fh Support for Error Translation by Network Redirectors

Network Error Translator Function 00h—Availability Check

INPUT:

AX	0500h

OUTPUT:

al	FFh	if redirector offers an error translation facility

Network Error Translator Function 01h—Translate Error

INPUT:

AX	0501h
BX	error code

SUCCESS:

cf	clear	
es:di	address of message as null-terminated string	
al	00h	if message requires completion, e.g., device name, drive letter
	01h	if message is complete

FAILURE:

cf	set
ax	corrupt
es:di	corrupt

CORRUPTS:

flags	in particular, interrupts are always enabled

The behavior described above is taken from the DOS 5 version of REDIR.EXE (the Microsoft Networks redirector) and IFSFUNC.EXE (distributed with DOS 4). Both programs expect to be the first to hook int 2Fh functions 05h and 11h and in that sense provide a standard definition for these interfaces. Be aware, however, that other redirectors—such as MSCDEX.EXE (which supplies DOS with knowledge about the CD-ROM file systems)—hook the int 2Fh chain to translate their own errors. Since the interface is undocumented, it is not clear whether rules exist about register preservation, so for safety, it is probably wise to assume that registers in which values are not returned explicitly are liable to corruption.

Subject to a few exceptions in versions before DOS 5, the error translation function is in fact reached by any value of AL in the range 00h < AL < F8h. If AL ≥ F8h, the function simply returns.

The DOS 4 IFSFUNC program offers two error translation functions: The 01h given above and a function 02h. In both cases, the IFSFUNC layer passes the error code to the relevant IFS driver for translation. If the IFS driver does not provide a translation, then ordinarily, IFSFUNC attempts its own interpretation of the error code, but if the function number is 02h, it returns the failure.

In DOS 3.30 and presumably earlier, the (non-zero) error code is passed in AL instead of BX.

For reference, error codes for which translations are available via int 2Fh function 05h, as offered by REDIR and MSCDEX, are listed in the following tables:

Table 17-5: Network Error Codes and Messages Defined by REDIR.EXE

ERROR	DESCRIPTION
0032h	NET809: Network request not supported
0033h	NET801: Remote computer not listening
0034h	NET802: Duplicate name on network
0035h	NET803: Network path not found
0036h	NET804: Network busy
0037h	NET805: Network device no longer exists
0038h	NET806: Net BIOS command limit exceeded
0039h	NET807: Network adapter hardware error
003Ah	NET808: Incorrect response from network
003Bh	NET810: Unexpected network error
003Ch	NET811: Incompatible remote adapter
003Dh	NET812: Print queue full
003Eh	NET813: Queue not full
003Fh	NET814: Not enough space for print file

Table 17-5: Network Error Codes and Messages Defined by REDIR.EXE

ERROR	DESCRIPTION
0040h	NET815: Network name was deleted
0041h	Access denied
0042h	NET817: Network device type incorrect
0043h	NET818: Network name not found
0044h	NET819: Network name limit exceeded
0045h	NET820: Net BIOS session limit exceeded
0046h	NET821: Sharing temporarily paused
0047h	NET823: Network request not accepted
0048h	NET822: Print or disk redirection is paused
0049h	NET476: Network software not installed
004Ah	NET477: Unexpected adapter close
0058h	Network data fault

Table 17-6: Error Codes and Messages Defined by MSCDEX.EXE

ERROR	DESCRIPTION
0064h	CDR100: Unknown error
0065h	CDR101: Not ready
0066h	CDR102: EMS memory no longer valid
0067h	CDR103: CDROM not High Sierra or ISO-9660 format
0068h	CDR104: Door open

Extended Error Information

Many of the DOS error codes listed in tables throughout the preceding section can never be returned directly by an int 21h function to indicate an error, that is, in register ax with the carry flag set.

For backward compatibility, DOS usually maps the error code into a smaller range. History being what it is, the natural code is referred to as an "extended" error.

Sometimes the full error information is discarded before reaching the mapping routine. For instance, once the read/write functions actually attempt to access a file on disk, having established that the handle is valid, and so forth, all error conditions are reduced to 0005h.

Table 17-7: Error Codes Returned in ax by DOS Functions

FUNCTION	POSSIBLE ERROR CODES IN AX
	Codes marked * can be returned in ax by the corresponding DOS function only in version 4.0 and higher. For each function, an extended error code that is not in the list of permissible return values causes ax to be given the value in bold type.
38h	0001h, **0002h**
39h	0002h, 0003h, **0005h**
3Ah	0002h, 0003h, **0005h**, 0010h
3Bh	0002h, **0003h**
3Ch	0002h, 0003h, 0004h, **0005h**, 0057h*
3Dh	0002h, 0003h, 0004h, **0005h**, 000Ch, 001Ah*, 0057h*
3Eh	**0006h**
3Fh	**0005h**, 0006h
40h	**0005h**, 0006h
41h	0002h, 0003h, **0005h**
42h	**0001h**, 0006h
43h	0001h, 0002h, 0003h, **0005h**
44h	0001h, **0005h**, 0006h, 000Dh, 000Fh
45h	**0004h**, 0006h
46h	**0004h**, 0006h
47h	**000Fh**, 001Ah*
48h	0007h, **0008h**
49h	0007h, **0009h**
4Ah	0007h, **0008h**, 0009h
4Bh	0001h, 0002h, 0003h, 0004h, **0005h**, 0008h, 000Ah, 000Bh
4Eh	0002h, 0003h, **0012h**
4Fh	**0012h**
56h	0002h, 0003h, **0005h**, 0010h*, 0011h
57h	**0001h**, 0006h, 0008h*, 000Dh*
58h	**0001h**
5Ah	0002h, 0003h, 0004h, **0005h**, 0057h*

Table 17-7: Continued

ERROR	DESCRIPTION
5Bh	0002h, 0003h, 0004h, **0005h**, 0050h, 0057h*
5Ch	0001h, 0006h, **0021h**, 0024h
65h	0001h, **0002h**
66h	0001h, **0002h**, 0041h*
67h	**0001h**, 0004h, 0008h
68h	**0006h**
69h	0001h, **0005h**, 000Dh, 000Fh
6Ch	0001h, 0002h, 0003h, 0004h, **0005h**, 0008h, 000Ch, 000Dh, 001Ah, 0050h, 0

Functions that did not exist before DOS version 3.00 are free to return values in ax that cannot be returned by functions for which backward compatibility may be a problem. Lower functions may be called legitimately by software written for DOS 2.xx and in principle should not return with ax higher than 0012h—the highest value possible before DOS 3.00. Note however, that this principle was violated by version 4, whose additions are retained in DOS 5.

A similar effect attends the error code passed to critical error handlers in register DI. In very early versions of DOS, device driver error codes were undefined above 000Ch (General Failure). To this day, the DOS kernel truncates higher values that arise during native file processing. Networks may be able to arrange higher error codes when invoking an int 24h exception through int 2Fh function 1206h.

Retrieving Extended Error Information

In practice, programs that encounter an error condition from one int 21h function often issue another DOS call to obtain extended information to clarify the problem:

Table 17-8: DOS function 59h—Get Extended Error Information

INPUT:

AH	59h

OUTPUT:

cf	clear
ax	extended error code
bl	suggested action

Table 17-8: Continued

bh	error class
ch	error location
cl	corrupt
es:di	if extended error code is Invalid Disk Change: address of structure giving media ID for "valid" disk

OFFSET	SIZE	DESCRIPTION
00h	0Ch	volume label as ASCIIZ string
0Ch	dword	serial number

otherwise: corrupt

According to the DOS *Programmer's Reference*, DOS alters all registers (cl, dx, si, bp, and ds, in addition to those in which values are returned explicitly) when processing this function. Although this is not true in versions of DOS released to date, programmers should heed the caution.

This function is processed on the third internal DOS stack—the one normally used to process int 21h functions 01h to 0Ch when a critical error is pending. This means that function 59h may be issued safely by an int 24h handler.

The address returned as es:di and the media ID found there are meaningful only if the extended error code indicates an Invalid Disk Change. As an extended note about this DOS function, observe that although the official reference assigns no meaning to the address returned in es:di, the default int 24h handler provided by COMMAND.COM uses the address to advise which disk should be re-inserted to recover from an error condition caused by an unexpected disk change. Much has been said about the ability of DOS to protect users against data corruption resulting from unexpected disk changes. The official reference describes accurately how a device driver that refuses a read or write command because of an Invalid Disk Change is expected to also return the volume label of the disk it expected, but without also documenting a way to retrieve this volume label from the DOS kernel, this ability to ask for the expected disk by name is kept from programmers writing their own critical error handlers.

To be precise, the extended error information does not describe the most recent DOS call, but rather the most recent error condition reported outside the DOS kernel. This is perhaps a subtle distinction. One of the consequences is hinted at by the observation that DOS function 59h can be called by a critical error handler. Before issuing an int 24h exception, DOS ensures that extended error information is available to describe the error condition in more detail.

Preserving Extended Error Information—a Note for TSRs

When extended error information is sought through one DOS call to clarify an error condition generated by an earlier DOS call, there arises a problem of integrity in a multitasking environment. Between the two DOS calls, an interrupting process is free in principle to issue int 21h instructions itself, since it will not be subverting any int 21h activity. A duty exists, however, to preserve the extended error information that the interrupted process may well expect to retrieve. It is well known, therefore, that apart from the handful of re-entrant DOS functions (int 21h functions 33h, 50h, 51h, 62h, and 64h), *no* int 21h functions should be used by a pop-up routine without first calling int 21h function 59h (Get Extended Error Information) to save the existing error information and that the last DOS call issued by the interrupting process (again with the exception of the reentrant functions) must be int 21h function 5D0Ah (Set Extended Error Information) to restore the extended error information as if there had been no intervention.

Table 17-9: DOS function 5D0Ah—Set Extended Error Information

INPUT:

AX	5D0Ah
DS:DX	address of parameter block

OFFSET	SIZE	DESCRIPTION
00h	word	extended error code
02h	byte	suggested action
03h	byte	error class
05h	byte	error location
0Ah	word	offset component of volume label address
0Eh	word	segment component of volume label address
12h	word	machine ID—field ignored if Windows enhanced mode is running
14h	word	process ID

OUTPUT:

al	corrupt

The parameter block has the format used for other (undocumented) subfunctions of int 21h function 5Dh. In general, each field corresponds to a register:

OFFSET	SIZE	REGISTER
00h	word	ax

02h	word	bx
04h	word	cx
06h	word	dx
08h	word	si
0Ah	word	di
0Ch	word	ds
0Eh	word	es
10h	word	unused
12h	word	machine ID—field ignored if Windows enhanced mode is running
14h	word	process ID

In this context, the positions required for the individual pieces of extended error information match the registers in which DOS function 59h returns them.

CHAPTER 18

Duplicating Disks via Extended Memory

This final chapter puts everything together in a DISKDUP program that uses XMS extended memory to help duplicate disks.

Because of functionality encapsulated in other modules, the top-level module for DISKDUP is fairly simple. As shown below, it calls `GetDiskCharacteristics`, `GetDiskMediaID`, and `ReadDisk` on the source diskette, then goes into a loop calling `WriteDisk` for each target diskette. These functions appear in DISKIO.C and DISKID.C, also shown in this chapter.

```
/* main.c - primary module for diskdup.exe */

#include    <conio.h>
#include    <dos.h>
#include    <malloc.h>
#include    <stdarg.h>
#include    <stdio.h>
#include    <stdlib.h>
#include    <string.h>

#include    <doserror.h>
#include    <drivparm.h>
#include    <standard.h>

static CHAR description [] = "\n\
Duplicates a floppy disk in the same drive, using conventional\n\
and extended memory to avoid tedious disk changes.\n\
\n\
  DISKDUP [drive:]\n\
";

/*  The program uses int 21h function 440Dh services to access storage media
    at a level below the file system. Details are handled externally (DISKIO.C
    and DISKID.C) but this module must supply some relevant structures
    and flags to keep track of the relationship between source and
    destination disks. */

extern BOOL GetDiskCharacteristics (WORD, struct DRIVPARM *); /*  diskid.c  */
extern BOOL GetDiskMediaID (WORD, struct MEDIAID *, BOOL *); /*  diskid.c  */
extern BOOL ReadDisk (WORD, struct DRIVPARM *, void *);  /*  diskio.c  */
extern BOOL WriteDisk (WORD, struct DRIVPARM *, void *); /*  diskio.c  */

struct MEDIAID SrcID, DestID;
BOOL bSrcID, bDestID;

struct DRIVPARM SrcParm, DestParm;

BOOL bSetDrivparm = TRUE;

void DrainStdin (void);
BOOL GetYesNo (void);
void PromptForKey (void);
void PutError (CHAR *, ...);
void PutNewLine (void);
BOOL RetryError (CHAR *, ...);

/*  =====================================================================  */

int main (int argc, char **argv)
{
    register CHAR *ptr;
    WORD drive, driveletter, tracksize;
    BOOL drivegiven = FALSE;
    void *buffer;

    /*   Parse the command line. */

    while (++ argv, -- argc) {
         ptr = *argv;
         if (*ptr == '/') {
             ptr ++;

             /*   A switch is provided to demonstrate a shortcoming of the
                  official documentation for Generic IOCTL services—refer to
                  the accompanying text for details. Any initial substring is
                  accepted, irrespective of case. */

             if (strnicmp ("SetDrivparmOff", ptr, strlen (ptr)) == 0) {
                  bSetDrivparm = FALSE;
```

```
            continue;
        }

        /*  A description of the program may be obtained by supplying
            the /? switch as usual. */

        if (ptr [0] == '?' AND ptr [1] == '\0') {
            printf (description);
            return (0);
        }

        /*  No other switches are accepted. Exit codes, incidentally,
            conform loosely to those of DISKCOPY. */

        PutError ("invalid switch—%s", -- ptr);
        return (4);
    }
    else {

        /*  The drive containing the disk to be duplicated may be given
            in upper or lower case, but as usual, must be followed
            immediately by a colon and nothing else. Do not allow the
            user to specify two drives. Record the drive as a 0-based
            number. */

        if (drivegiven) {
            PutError ("too many arguments");
            return (4);
        }
        driveletter = ptr [0] & ~0x20;
        drive = driveletter—'A';
        if (drive <= 'Z'—'A' AND ptr [1] == ':' AND ptr [2] == '\0') {
            drivegiven = TRUE;
            continue;
        }

        /*  No other parameters are accepted. */
        PutError ("invalid parameter—%s", ptr);
        return (4);
    }
}

/*  If no drive was specified on the command line, call DOS to determine
    the default drive. Most int 21h functions work with 1-based drive
    numbers. In this program, drive numbers are kept in 0-based form
```

```
          except when calling DOS functions (or direct C counterparts in the
          _dos style of the Microsoft C library). */

     if (NOT drivegiven) {
          _dos_getdrive (&drive);
          drive --;
          driveletter = drive + 'A';
     }

     /*   Determine the nature of the disk in the nominated drive, possibly
          also prompting for a disk or displaying a fatal error message if the
          disk or drive is unacceptable. Those choices however are hidden
          from this level of the program's modular heirarchy. */

     if (NOT GetDiskCharacteristics (drive, &SrcParm)) return (4);

     /*   Reject attempts to duplicate fixed disks (a design decision rather
          than a lower-level error). */

     if (SrcParm.wDeviceAttributes & DRIVE_FIXED) {
          PutError ("drive %c: contains nonremovable media", driveletter);
          return (4);
     }

     /*   While still investigating the source disk's properties, record its
          volume label and serial number. The GIOCTL interface works beneath
          the DOS file system and the program is therefore vulnerable to
          unanticipated disk changes (as is the standard DISKCOPY utility).
          The media identifiers, particularly the serial number introduced
          with DOS 4, may help the program trap accidents or silly behavior. */

     if (NOT GetDiskMediaID (drive, &SrcID, &bSrcID)) return (4);

     /*   Allocate a buffer from the heap to hold successive tracks read from
          disk before they are passed to more substantial storage. The same
          buffer will be used when the process is reversed to make the
          duplicate disk. */

     tracksize = SrcParm.bpb.wBytesPerSector * SrcParm.bpb.wSectorsPerTrack;
     buffer = malloc (tracksize);
     if (buffer == NULL) {
          PutError ("insufficient conventional memory");
          return (4);
     }
```

```
/*   Read the disk a track at a time, copying each track into whatever
     temporary storage is available--whether this be conventional memory,
     extended memory or some file on a hard disk, or a combination of
     these (the details are of no concern at this level). */

printf ("\nReading disk, please wait ...\n");
if (NOT ReadDisk (drive, &SrcParm, buffer)) return (3);

/*   Ask for a second disk, then obtain its characteristics for
     comparison with those of the source. */

for (;;) {
     printf ("\n\nInsert target disk in drive %c:", driveletter);
     for (;;) {
         PromptForKey ();
         if (NOT GetDiskCharacteristics (drive, &DestParm)) return (4);

         /*    In this simple implementation, the target disk must already
               be formatted to the same specifications as the source disk.
               Should this not be so, alert the user. Further in the for
               loop there is a message offering the user the chance to
               replace the disk. */

         if (memcmp (&DestParm.bpb, &SrcParm.bpb, sizeof (struct BPB))) {
               printf ("\nTarget disk must have same format as source.");
         }
         else {

               /*  If the source and target disks do have the same format,
                   guard against the possibility that they are the same
                   disk—that the user has not answered the prompt for a
                   new disk without actually having changed disks. */

               if (NOT GetDiskMediaID (drive, &DestID, &bDestID))
                  return (4);

               if (NOT (bDestID AND bSrcID)) break;
               if (memcmp (&DestID, &SrcID, sizeof (SrcID)) != 0) break;

               printf ("\nTarget disk has same serial number as source.");
               printf ("\nIs this disk intended as the target (y/n)?  ");

               if (GetYesNo ()) {
                   PutNewLine ();
                   break;
```

```
        }
                PutNewLine ();
            }
            printf ("\nInsert replacement disk in drive %c:", driveletter);
        }

        /*  Retrieve the contents of the source disk from storage and copy
            them to the new disk in the drive (again, the details are of no
            concern at this level). */

        printf ("\nWriting disk, please wait ...\n");
        if (WriteDisk (drive, &DestParm, buffer)) {
            printf ("\n\nDisk duplicated ok. Make another copy (y/n)?  ");
            if (NOT GetYesNo ()) {
                PutNewLine ();
                return (0);
            }
        }
        else {
            printf ("\n\nDuplication failure. Try another disk (y/n)?  ");
            if (NOT GetYesNo ()) {
                PutNewLine ();
                return (3);
            }
        }
    }
}

/*  --------------------------------------------------------------------  */

/*  The remaining functions provide uniform treatment of console I/O for
    prompts and error messages. */

void DrainStdin (void)
{
    while (khit ()) {
        getch ();
    }
}

BOOL GetYesNo (void)
{
    WORD ch;

    for (;;) {
        ch = getch ();

        /*  Even for something as simple as waiting for a y/n response,
            remember that special function keys, Alt combinations, etc. are
            signalled by two bytes—if the first byte is null, do not leave
            the second in the console buffer to confuse other routines or
            programs. */

        if (ch == 0) {
            getch ();
            continue;
        }
        if (ch == 'y'OR ch == 'Y') {
            putch (ch);
            return (TRUE);
        }
        if (ch == 'n' OR ch == 'N') {
            putch (ch);
            return (FALSE);
        }
    }
}

void PromptForKey (void)
{
    printf ("\nPress any key when ready ...\n");
    DrainStdin ();
    while (getch () == 0) {
    }
}

void PutError (CHAR *str, ...)
{
    va_list va_start (argptr, str);
    printf ("\nDISKDUP fatal error:  ");
    vprintf (str, argptr);
    PutNewLine ();
    va_end (argptr);
}

void PutNewLine (void)
{
    printf ("\n");
}
```

```
BOOL RetryError (CHAR *str, ...)
{
    register WORD answer;
    va_list va_start (argptr, str);

    vprintf (str, argptr);
    va_end (argptr);
    printf ("\nRetry (y/n)?  ");
    answer = GetYesNo ();
    PutNewLine ();
    return (answer);
}
```

It is important to stress that the DISKDUP program is for instructional purposes only and therefore omits what would be a requirement in a commercial version, that is, to *format* the target disk. As it stands now, each target diskette must already be formatted in the same way as the source diskette.

One important consequence is that after the DISKDUP program gets the device parameters (IOCTL minor code 60h), it has no obvious need to set the device parameters. After all, it's not intending to change them! Nonetheless, DISKDUP does set the device parameters (unless run with the **/s** or **/SetDrivparmOff** switch). Nothing in the documentation would suggest the need to set device parameters if they're already what you want, but if you don't, then very strange things happen.

If you reboot and run **diskdup /s** on a drive that hasn't been accessed, then it can take seven minutes (count 'em) to read a 1.2MB disk, for the simple reason that nobody has told IO.SYS that the disk has good (i.e., contiguous) track layout (a property you can't determine though any DOS interface). If you run a program that does set the device parameters, then afterward, **diskdup /s** takes the normal time.

The **/s** switch is provided to show what would happen if you didn't set the device parameters, say, because your particular use of the disk didn't involve changing the parameters you had just got from the driver. A programmer relying on the documentation could all too easily not see a need to set the device parameters in certain situations. If he doesn't, then his application will suffer drastically in performance whenever it operates on a drive that has not yet been used in the DOS session. Even if your immediate needs do not require you to set the device parameters before using the various Generic IOCTL services for disk access, you must do so. Otherwise, the default block device driver will be reading and writing each sector one at a time (because it suspects they may vary in size throughout a track, therefore making it unsafe to just read/write the whole track in one go).

There is another problem that DISKDUP tries to address but which is not strictly possible to resolve from DOS. How do you stop users from changing disks in the middle of a DISKDUP? DISKDUP deals pretty effectively with the problem outside of task-switching by checking the media ID every time it judges that the user has had a chance to be nasty (note the calls to `HasDiskChanged` within `WriteTrack` and `ReadTrack`).

DISKDUP could, of course, refuse to run if it notices a task switcher to be present, but that would be silly. What DOS really ought to provide is a way for applications to allocate use of a disk drive. Then DISKDUP could say, "Okay, I want drive A and if anybody else uses it, then either give them an invalid disk change error or take full responsibility for ensuring that the disk is restored."

Disk Identification

DISKID.C provides the DISKDUP utility with functions that deal with disk identification. It provides the `GetDiskCharacteristics` and `GetDiskMediaID` functions used in MAIN.C. It also provides the `HasDiskChanged` function, used to deal with unanticipated disk changes. DISKID.C uses the generic IOCTL functions from GIOCTL.C in Chapter 16.

```
/* diskid.c */

#include    <dos.h>
#include    <stdio.h>
#include    <string.h>

#include    <doserror.h>
#include    <drivparm.h>
#include    <standard.h>

BOOL GetDiskCharacteristics (WORD, struct DRIVPARM *);
BOOL GetDiskMediaID (WORD, struct MEDIAID *, BOOL *);
BOOL HasDiskChanged (WORD drive, struct MEDIAID *idptr);
BOOL SetDiskMediaID (WORD, struct MEDIAID *, BOOL *);

/*  Use int 21h function 440Dh to access storage media below the level of
    the file system. This allows DISKDUP to work with local drives that might
    not be supported by the int 13h BIOS services. The following C functions
    each correspond to an int 21h call and follow the Microsoft C scheme of
    returning the DOS error code on failure or zero on success. Like most
    int 21h functions, these expect 1-based drive numbers. */

extern WORD _dos_getdrivparm (WORD, struct DRIVPARM *);     /*  gioctl.c  */
extern WORD _dos_getmediaid (WORD, struct MEDIAID *);       /*  gioctl.c  */
extern WORD _dos_setmediaid (WORD, struct MEDIAID *);       /*  gioctl.c  */

/*  A function to describe DOS error values in a standard form  */

extern CHAR *GetErrormsg (WORD);          /*  errormsg.c  */
```

```
/*  Some self-explanatory functions declared in MAIN.C  */

extern void PromptForKey (void);
extern void PutError (CHAR *, ...);
extern BOOL RetryError (CHAR *, ...);

/* ======================================================================== */

/*  The following function fills a DRIVPARM structure about the current
    contents of a designated drive. The function prompts for a disk, if the
    drive is empty. */

BOOL GetDiskCharacteristics (WORD drive, struct DRIVPARM *parmptr)
{
    register WORD driveletter, errorcode;
    struct DOSERROR errorinfo;

    driveletter = drive + 'A';

    /*  Note that any attempt at obtaining disk characteristics may have
    to be repeated if the drive is empty, hence the for loop. */

    for (;;) {
    parmptr -> cSpecial = DRIVPARM_CURRENT;
    if (_dos_getdrivparm (drive + 1, parmptr) == 0) return (TRUE);

    /*  The normal DOS error code returned by the Generic IOCTL function
        is not sufficiently informative - note in particular that the
        value 0x0005 covers all device errors, whatever their cause - so
        obtain extended error information. */

    errorcode = dosexterr (&errorinfo);

    /*  If disk characteristics cannot be obtained because the drive
        is undefined or remote or because the responsible device driver
        does not offer a Generic IOCTL interface, deal with the error
        here. */

    if (errorcode == E_INVALID_DRIVE) {
        PutError ("drive %c: is invalid", driveletter);
        return (FALSE);
    }
    if (errorcode == E_INVALID_FUNCTION OR errorcode == E_BAD_COMMAND) {
        PutError ("drive %c: is unsuitable", driveletter);
        return (FALSE);
    }

    /*  If the problem is simply that the drive contains no disk, prompt
        for one and try again. Strictly speaking, the user should not
        be asked to insert a disk until the program has first checked
        that media replacement is possible. */

    if (errorcode == E_NOT_READY) {
        printf ("\nInsert disk in drive %c:", driveletter);
        PromptForKey ();
        continue;
    }

    /*  For any other problem, describe the error in some detail and
        offer the chance to retry. Note that device errors during IOCTL
        services do not generate int 24h exceptions - the message
        displayed by the RetryError function is not too dissimilar
        however to the familiar "Abort, Retry, Ignore, Fail". */

    if (NOT RetryError ("%s reading drive %c:",
        GetErrormsg (errorcode), driveletter)) return (FALSE);
    }
}

/* ------------------------------------------------------------------------ */

/*  Two functions are presented for dealing with media identifiers (volume
    label, serial number and file system). Both functions pass the 0-based
    drive number and two addresses: the first points to a MEDIAID
    structure whose validity is indicated by the Boolean flag at the second
    address.

    The functions return FALSE if a device driver error prevents
    determination of the existence of a media id for the disk in the
    designated drive or of the ability of the disk format to accept a new
    media id. */

BOOL GetDiskMediaID (WORD drive, register struct MEDIAID *idptr, BOOL *flagptr)
{
    struct DOSERROR errorinfo;
    WORD errorcode;

    for (;;) {
    idptr -> wInfoLevel = 0;
    if (_dos_getmediaid (drive + 1, idptr) == 0) {
        *flagptr = TRUE;
        return (TRUE);
    }
    *flagptr = FALSE;

    errorcode = dosexterr (&errorinfo);
    if (errorcode == E_INVALID_FUNCTION OR errorcode == E_NOT_DOS_DISK)
        return (TRUE);

    if (NOT RetryError ("%s reading media id drive %c:",
        GetErrormsg (errorcode), drive + 'A')) return (FALSE);
    }
}

BOOL SetDiskMediaID (WORD drive, register struct MEDIAID *idptr, BOOL *flagptr)
{
    struct dosdate_t date;
    struct dostime_t time;
    struct DOSERROR errorinfo;
    WORD errorcode;
    DWORD jumbledtime;

    for (;;) {

    /*  This function adds a 'unique' serial number to the other media
        identifiers, sparing the caller from the details of concocting
        something suitable. Microsoft's choice of structures for the
        C library routines that give access to the DOS Get Date/Time
        functions do not lend themselves immediately to creating a
        serial number compatible with the standard FORMAT program.
        That utility simply adds the cx values returned by the two
        functions to get the high word of the new serial number, then
        does the same with the dx values to get the low word. The same
        can be achieved easily enough here with a little jumbling. */

    _dos_getdate (&date);
    _dos_gettime (&time);
    jumbledtime = MK_DWORD (MK_WORD (time.hour, time.minute),
            MK_WORD (time.second, time.hsecond));
    idptr -> dwSerialNumber = *((DWORD *) &date) + jumbledtime;
    idptr -> wInfoLevel = 0;
    if (_dos_setmediaid (drive + 1, idptr) == 0) {
        *flagptr = TRUE;
        return (TRUE);
    }
    *flagptr = FALSE;

    /*  In general, alert the user to an error, but note that there is
        not much point to this if the problem lies with a disk format
        that doesn't support the media identifiers. */

    errorcode = dosexterr (&errorinfo);
    if (errorcode == E_INVALID_FUNCTION OR errorcode == E_NOT_DOS_DISK)
        return (TRUE);

    if (NOT RetryError ("%s writing media id drive %c:",
        GetErrormsg (errorcode), drive + 'A')) return (FALSE);
    }
}

/* ---------------------------------------------------------------------- */

BOOL HasDiskChanged (WORD drive, register struct MEDIAID *idptr)
{
    struct MEDIAID currentid;
    BOOL flag;

    /*  The argument idptr (if not NULL) addresses a structure containing
    the media identifiers expected of the disk in the specified drive.
    Comparison with the identifiers belonging to the disk that actually
    is in the drive may show that the drive contents have changed (in
    which case the function returns TRUE).

    In the first instance however, return FALSE (meaning don't know) if
    some sort of disk error prevents determination even of the existence
    of media identifiers. */

    if (NOT GetDiskMediaID (drive, &currentid, &flag)) return (FALSE);

    /*  If the current contents of the drive do not offer a media id, give
    benefit of doubt if identifiers were not expected, but otherwise
    regard the disk as changed. */

    if (NOT flag) return (idptr == NULL ? FALSE : TRUE);

    /*  Given that media identifiers exist now, regard the disk as changed
    if there were no identifiers expected or if the current identifiers
    differ from the expected ones. */

    if (idptr == NULL) return (TRUE);
    if (memcmp (&currentid, idptr, sizeof (*idptr))) return (TRUE);

    /* That the current contents have the expected media id does not prove
    that no disk change has occurred. */

    return (FALSE);
}
```

Disk Access

DISKIO.C contains the high-level `ReadDisk` and `WriteDisk` functions, which are called by the main DISKDUP module. These functions in turn call the `ReadTrack` and `WriteTrack` functions, also in DISKIO.C. As noted earlier, `WriteDisk` also must call `SetDrivparm` even though DISKDUP doesn't change disk parameters.

```
/* diskio.c */

#include    <dos.h>
#include    <stdio.h>
#include    <string.h>

#include    <doserror.h>
#include    <drivparm.h>
#include    <standard.h>

/* This file provides the DISKDUP utility with functions that arrange disk
    access. */

BOOL ReadDisk (WORD, struct DRIVPARM *, void *);
BOOL WriteDisk (WORD, struct DRIVPARM *, void *);

static BOOL ReadTrack (WORD, WORD, register struct RWBLOCK *);
static BOOL WriteTrack (WORD, WORD, register struct RWBLOCK *);
static void SetDrivparm (WORD, struct DRIVPARM *);

extern BOOL GetDiskMediaID (WORD, struct MEDIAID *, BOOL *);
extern BOOL HasDiskChanged (WORD drive, struct MEDIAID *idptr);
extern BOOL SetDiskMediaID (WORD, struct MEDIAID *, BOOL *);

extern WORD _dos_getdrivparm (WORD, struct DRIVPARM *);      /*    gioctl.c   */
extern WORD _dos_setdrivparm (WORD, struct DRIVPARM *);      /*    gioctl.c   */
extern WORD _dos_readtrack (WORD, struct RWBLOCK *);         /*    gioctl.c   */
extern WORD _dos_writetrack (WORD, struct RWBLOCK *);        /*    gioctl.c   */

extern BOOL bSetDrivparm;                                    /*    main.c     */
extern struct MEDIAID SrcID, DestID;                         /*    main.c   */
extern BOOL bSrcID, bDestID;                                 /*    main.c   */

static BOOL ChangeDestID (WORD, struct MEDIAID *, BOOL);
static BOOL bDestIDChanged;

/*  Questions about where the material read from the source disk is held
    until the target disk is ready are all dealt with externally. This
    module need know only a few abstract operations. */

extern BOOL ArrangeStorage (DWORD);                          /*    storage.c  */
extern WORD ReadStorage (void _far *, WORD);                 /*    storage.c  */
extern WORD WriteStorage (void _far *, WORD);                /*    storage.c  */
extern BOOL RewindStorage (void);                            /*    storage.c  */

/*  A function to describe DOS error values in a standard form     */

extern CHAR *GetErrormsg (WORD);          /*    errormsg.c  */

/*  Some self-explanatory functions declared in MAIN.C   */

extern void PromptForKey (void);
extern void PutError (CHAR *, ...);
extern BOOL RetryError (CHAR *, ...);

/*  ========================================================================  */

/*  Although the program never alters the characteristics of a disk, it
    still requires the GIOCTL service that sets drive parameters. It isn't
    stated very clearly in the official literature, but for a number of
    reasons, mostly concerning undocumented track layout fields, the Generic
    IOCTL commands for reading and writing tracks, as implemented for the
    default block device driver in IO.SYS, are prone to peculiar behavior
    if the device parameters are not set first. Refer to the accompanying
    text for details. */

static void SetDrivparm (WORD drive, struct DRIVPARM *parmptr)
{
    register WORD sector;
    register struct SECTORPARM *secptr;

    /*   It is assumed here that the DRIVPARM structure already contains a
         BPB for the media and that the only problem is track layout. */

    parmptr -> cSpecial = SET_TRACKLAYOUT_ONLY | SET_GOOD_TRACKLAYOUT;
    #ifdef _MINIMAL_

    /*   A minimal solution is to assume that the default track layout table
         is still in place—that no operations are being conducted on disks
         with unusual sector sizes, let alone disks that have variations
         within a track. In this case, there is an easy way to have the
         Set Device Parameters command avoid loading a track layout table. */

    parmptr -> wSectorsInTable = 0;

    #else

    /*   For safety however, the track layout table should be refreshed. */

    parmptr -> wSectorsInTable = parmptr -> bpb.wSectorsPerTrack;

    secptr = parmptr -> tracklayout;
    sector = 1;
    do {
         secptr -> wNumber = sector;
         secptr -> wSize = 512;
         secptr ++;
         sector ++;
    } while (sector < parmptr -> wSectorsInTable);

    #endif

    _dos_setdrivparm (drive + 1, parmptr);
}

/*   ------------------------------------------------------------------------   */

BOOL ReadDisk (WORD drive, struct DRIVPARM *parmptr, void *buffer)
{
    register WORD bytes_per_track, heads;
    WORD cylinders, sectors_per_track;
    DWORD total_bytes, total_sectors;
    struct RWBLOCK rwblock;

    if (bSetDrivparm) SetDrivparm (drive, parmptr);

    /*   Calculate certain useful quantities. */

    heads = parmptr -> bpb.wHeads;
    sectors_per_track = parmptr -> bpb.wSectorsPerTrack;
    bytes_per_track = sectors_per_track * parmptr -> bpb.wBytesPerSector;
    total_sectors = parmptr -> bpb.wSectors ?
              (DWORD) parmptr -> bpb.wSectors : parmptr -> bpb.dwHugeSectors;
    cylinders = (WORD) (total_sectors / (sectors_per_track * heads));
    total_bytes = total_sectors * parmptr -> bpb.wBytesPerSector;

    /*    Prepare the necessary storage space, making sure that enough of it
          exists to hold the contents of the entire disk. */

    if (NOT ArrangeStorage (total_bytes)) {
          PutError ("insufficient temporary storage for entire disk");
          return (FALSE);
    }

    /*    Prepare the RWBLOCK structure for the Generic IOCTL command that
          reads disk tracks, then loop through the disk, track by track. */

    rwblock.cSpecial = 0;
    rwblock.wCylinder = 0;
    rwblock.wFirstSector = 0;
    rwblock.wSectors = sectors_per_track;
    rwblock.lpBuffer = (void _far *) buffer;

    do {
          rwblock.wHead = 0, heads = parmptr -> bpb.wHeads;
          do {
               if (NOT ReadTrack (drive, bytes_per_track, &rwblock))
                    return (FALSE);
          } while (++ rwblock.wHead, -- heads);
    } while (++ rwblock.wCylinder, -- cylinders);

    return (TRUE);
}

BOOL WriteDisk (WORD drive, struct DRIVPARM *parmptr, void *buffer)
{
    register WORD bytes_per_track, heads;
    WORD cylinders, sectors_per_track;
    DWORD total_bytes, total_sectors;
    struct RWBLOCK rwblock;

    if (bSetDrivparm) SetDrivparm (drive, parmptr);

    heads = parmptr -> bpb.wHeads;
    sectors_per_track = parmptr -> bpb.wSectorsPerTrack;
    bytes_per_track = sectors_per_track * parmptr -> bpb.wBytesPerSector;
    total_sectors = parmptr -> bpb.wSectors ?
             (DWORD) parmptr -> bpb.wSectors : parmptr -> bpb.dwHugeSectors;
    cylinders = (WORD) (total_sectors / (sectors_per_track * heads));
    total_bytes = total_sectors * parmptr -> bpb.wBytesPerSector;

    /*    The storage space is accessed sequentially—now that its contents
          must be retrieved, any internal position counters describing that
          storage will have to be rewound to the start. */

    if (NOT RewindStorage ()) {
          PutError ("error controlling temporary storage");
          return (FALSE);
    }

    bDestIDChanged = FALSE;

    rwblock.cSpecial = 0;
    rwblock.wCylinder = 0;
    rwblock.wFirstSector = 0;
    rwblock.wSectors = sectors_per_track;
    rwblock.lpBuffer = (void _far *) buffer;

    do {
          rwblock.wHead = 0, heads = parmptr -> bpb.wHeads;
          do {
              if (NOT WriteTrack (drive, bytes_per_track, &rwblock))
                    return (FALSE);
          } while (++ rwblock.wHead, -- heads);
    } while (++ rwblock.wCylinder, -- cylinders);

    return (TRUE);
}

/*  ------------------------------------------------------------------  */

static BOOL ReadTrack (WORD drive, WORD tracksize,
                                register struct RWBLOCK *rwptr)
{
    struct DOSERROR errorinfo;

    /*    Read the track indicated by fields in the RWBLOCK structure passed
          as one of the arguments. */

    while (_dos_readtrack (drive + 1, rwptr) != 0) {
        /*  If there is an error, alert the user and ask to retry the
            operation. */

        if (NOT RetryError ("%s reading track %d head %d",
                   GetErrormsg (dosexterr (&errorinfo)),
                   rwptr -> wCylinder, rwptr -> wHead))
            return (FALSE);

        /*  Before reattempting the track read, try to establish that the
            original source disk has not been swapped for another during
            the pause—there's no explaining what some people do with their
            computers!      */

        if (HasDiskChanged (drive, bSrcID ? &SrcID : NULL)) {
            PutError ("unexpected disk change");
            return (FALSE);
        }
    }

    /*  Having read the track successfully, append the data to the rest that
        has been read from disk, checking that the full amount is saved
        correctly. */

    if (WriteStorage (rwptr -> lpBuffer, tracksize) != tracksize) {
        PutError ("storage failure while saving track %d head %d",
            rwptr -> wCylinder, rwptr -> wHead);
        return (FALSE);
    }
    return (TRUE);
}

static BOOL WriteTrack (WORD drive, WORD tracksize,
                              register struct RWBLOCK *rwptr)
{
    struct DOSERROR errorinfo;
    struct MEDIAID id;
    BOOL idflag;

    /*  This is essentially the reverse of the previous function—but with
        some small exceptions. */

    if (ReadStorage (rwptr -> lpBuffer, tracksize) != tracksize) {
        PutError ("storage failure while retrieving track %d head %d",
            rwptr -> wCylinder, rwptr -> wHead);
        return (FALSE);
    }
    while (_dos_writetrack (drive + 1, rwptr) != 0) {
          if (NOT RetryError ("%s writing track %d head %d",
                    GetErrormsg (dosexterr (&errorinfo)),
                    rwptr -> wCylinder, rwptr -> wHead))
              return (FALSE);
          if (HasDiskChanged (drive, bDestID ? &DestID : NULL)) {
              PutError ("unexpected disk change");
              return (FALSE);
          }
    }

    /*    If either the source disk or the destination disk possessed media
          identifiers at the start, then at some stage during the copy
          operation, the destination disk's identifiers will change. Most
          probably, the change will occur when the first track is written,
          but this does not appear to be a formal requirement for media
          identifiers.

          The change has to be caught for two reasons:  first, to support
          detection of a disk change during an option to retry and second, to
          write a distinct serial number to the destination disk at the
          earliest possible opportunity. */

    if (NOT bDestIDChanged AND (bSrcID OR bDestID)) {
          if (NOT GetDiskMediaID (drive, &id, &idflag)) return (FALSE);
          bDestIDChanged = ChangeDestID (drive, &id, idflag);
    }

    return (TRUE);
}

static BOOL ChangeDestID (WORD drive, struct MEDIAID *idptr, BOOL idflag)
{
    if (idflag) {
          if (bSrcID) {

              /*    In this case, there is currently an id and there is
                    eventually meant to be (by the time the source disk has been
                    copied completely). If the destination disk had an id all
                    along and this differs from the source disk's, then the new
                    id has not been written to the destination yet. Return
                    FALSE so that the check will be repeated. */

              if (bDestID AND memcmp (idptr, &SrcID, sizeof (SrcID)))
```

```
                  return (FALSE);

            /*    Now it is clear that the destination disk has acquired the
                  id from the source. Update the DestID structure so that any
                  future disk changes will be caught and write a distinct
                  serial number to the destination disk. */

            memcpy (&DestID, idptr, sizeof (*idptr));
            return (SetDiskMediaID (drive, &DestID, &bDestID));
        }
        else {

            /*    For this case, there is currently an id on the destination
                  disk, but there should not be once the copy operation is
                  complete. Regard this situation as indicating that the
                  original media id on the destination disk has not yet been
                  overwritten. */

            return (FALSE);
        }
    }
    else {
        if (bSrcID) return (FALSE);
        else {
            bDestID = FALSE;
            return (TRUE);
        }
    }
}
```

Temporary Storage

Rather than directly access extended memory via XMS, the DISKDUP utility uses a general-purpose STORAGE module. The ReadTrack function in DISKIO.C calls `WriteStorage;` WriteTrack calls `ReadStorage`. The STORAGE facility is not limited to XMS memory; it has been set up to allow multiple forms of temporary storage. STORAGE.H defines the following interface:

```
struct STORAGE_MEDIUM {
    WORD flags;
    DWORD capacity;
    DWORD current;
    DWORD (*create) (DWORD);
    BOOL (*destroy) (void);
    WORD (*read) (void _far *, WORD);
    WORD (*write) (void _far *, WORD);
    BOOL (*rewind) (void);
};
```

Currently DISKDUP uses two modules that support this STORAGE interface: MEM.C and XMSFILE.C. Readers may also want to add a module that uses disk space in the TEMP directory.

```
/* storage.c */

#include    <dos.h>
#include    <signal.h>
#include    <stdio.h>
#include    <stdlib.h>

#include    <standard.h>

#include    "storage.h"

BOOL ArrangeStorage (DWORD);
WORD ReadStorage (void _far *, WORD);
WORD WriteStorage (void _far *, WORD);
BOOL RewindStorage (void);
void DestroyStorage (void);

extern struct STORAGE_MEDIUM MemStorage;  /* mem.c */
extern struct STORAGE_MEDIUM XMSStorage;  /* xmsfile.c */

static struct STORAGE_MEDIUM *storage [] = {&MemStorage, &XMSStorage};
#define STORAGE_MEDIA  (sizeof (storage) / sizeof (storage [0]))

static void CtrlCtrap (int);
static int _far harderrortrap (void);

static BOOL traps_set = FALSE;

extern void PutError (char *, ...);

/* ====================================================================

BOOL ArrangeStorage (DWORD requirement)
{
    register struct STORAGE_MEDIUM **smptr;
    register struct STORAGE_MEDIUM *ptr;
    WORD n;
    BOOL result = FALSE;

    if (requirement == 0) return (FALSE);

    if (NOT traps_set) {
        atexit (DestroyStorage);
        signal (SIGINT, CtrlCtrap);
        _harderr ((void (_far *) ()) (void CODESEG *) harderrortrap);
        traps_set = TRUE;
    }

    smptr = storage, n = STORAGE_MEDIA;
    do {
                ptr = *smptr ++;
                ptr -> capacity = ptr -> create (requirement);
                if (ptr -> capacity == 0) continue;

                ptr -> flags |= SELECTED;

                requirement -= ptr -> capacity;

                if (requirement == 0) {
                          result = TRUE;
                          break;
                }
    } while (-- n);

    while (-- smptr >= storage) {
        ptr = *smptr;
        if (NOT (ptr -> flags & SELECTED)) ptr -> destroy ();
    }

    return (result);
}

WORD WriteStorage (void _far *buffer, WORD count)
{
    register struct STORAGE_MEDIUM **smptr;
    register struct STORAGE_MEDIUM *ptr;
    WORD n;
    WORD numwrt, part_count;
    DWORD space;

    numwrt = 0;
    if (count == 0) return (numwrt);
    smptr = storage, n = STORAGE_MEDIA;
    do {
    ptr = *smptr ++;
    if (ptr -> flags & SELECTED AND NOT (ptr -> flags & FILLED)) {
        space = ptr -> capacity - ptr -> current;
        part_count = space < count ? (WORD) space : count;

        if (ptr -> write (buffer, part_count) != part_count)
        return (FALSE);

        ptr -> current += part_count;
        if (ptr -> current >= ptr -> capacity) ptr -> flags |= FILLED;

        (BYTE _far *) buffer += part_count;
        numwrt += part_count;
        count -= part_count;

        if (count == 0) break;
        }
    } while (-- n);

    return (numwrt);
}

BOOL RewindStorage (void)
{
    register struct STORAGE_MEDIUM **smptr;
    register struct STORAGE_MEDIUM *ptr;
    WORD n;

    smptr = storage, n = STORAGE_MEDIA;
    do {
    ptr = *smptr ++;
    if (ptr -> flags & SELECTED) {
        if (NOT (ptr -> rewind ())) return (FALSE);
        ptr -> flags &= ~RETRIEVED;
        ptr -> current = 0;
        }
    } while (-- n);

    return (TRUE);
}

WORD ReadStorage (void _far *buffer, WORD count)
{
    register struct STORAGE_MEDIUM **smptr;
    register struct STORAGE_MEDIUM *ptr;
    WORD n;
    WORD numread, part_count;
    DWORD space;

    numread = 0;
    if (count == 0) return (numread);

    smptr = storage, n = STORAGE_MEDIA;
    do {
    ptr = *smptr ++;
    if (ptr -> flags & SELECTED AND NOT (ptr -> flags & RETRIEVED)) {
        space = ptr -> capacity - ptr -> current;
        part_count = space < count ? (WORD) space : count;

        if (ptr -> read (buffer, part_count) != part_count) return (FALSE);

        ptr -> current += part_count;
        if (ptr -> current >= ptr -> capacity) ptr -> flags |= RETRIEVED;

        (BYTE _far *) buffer += part_count;
        numread += part_count;
        count -= part_count;

        if (count == 0) break;
        }
    } while (-- n);

    return (numread);
}

void DestroyStorage (void)
{
    register struct STORAGE_MEDIUM **smptr;
    register struct STORAGE_MEDIUM *ptr;
    WORD n;

    smptr = storage, n = STORAGE_MEDIA;
    do {
        ptr = *smptr ++;
        ptr -> destroy ();
    } while (-- n);
}
```

```
static void CtrlCtrap (int sigint)
{
    signal (SIGINT, SIG_IGN);
    PutError ("terminated by user");
    exit (2);
}

static int _far harderrortrap (void)
{
    return (_HARDERR_FAIL);
}
```

Conventional Memory

MEM.C is a module that uses the STORAGE interface to provide DISKDUP with conventional memory.

```
/* mem.c */

#include    <dos.h>
#include    <string.h>

#include    <standard.h>

#include    "storage.h"

static DWORD CreateMem (DWORD);
static BOOL DestroyMem (void);
static WORD ReadMem (void _far *, WORD);
static WORD WriteMem (void _far *, WORD);
static BOOL RewindMem (void);

struct STORAGE_MEDIUM MemStorage =
    {0, 0, 0, CreateMem, DestroyMem, ReadMem, WriteMem, RewindMem};

static _segment mem_seg = 0;
static DWORD mem_ptr = 0, mem_end = 0, mem_size = 0;

static DWORD CreateMem (DWORD requirement)
{
    DWORD paras;
    WORD temp;

    paras = (requirement + 15) >> 4;
    if (HIGHWORD (paras)) paras = 0xFFFF;
    if (_dos_allocmem ((WORD) paras, &temp) == 0) mem_size = requirement;
    else {
    paras = temp;
    if (_dos_allocmem ((WORD) paras, &temp) == 0) mem_size = paras << 4;
    else return (0);
    }
    mem_seg = temp;
    mem_ptr = mem_end = 0;
    return (mem_size);
}

static BOOL DestroyMem (void)
{
    if (mem_seg) {
    _dos_freemem (mem_seg);
    mem_seg = 0;
    }
    return (TRUE);
}

static WORD ReadMem (void _far *buffer, WORD count)
{
    DWORD avail;
    BYTE _far *src;

    avail = mem_end - mem_ptr;
    if (avail < count) count = (WORD) avail;

    if (count) {
    src = (BYTE _huge *) MK_FP (mem_seg, NULL) + mem_ptr;
    _fmemcpy (buffer, src, count);
    mem_ptr += count;
    }
    return (count);
}

static WORD WriteMem (void _far *buffer, WORD count)
{
    DWORD avail;
    BYTE _far *dest;

    avail = mem_size - mem_ptr;
    if (avail < count) count = (WORD) avail;

    if (count) {
```

```
    dest = (BYTE _huge *) MK_FP (mem_seg, NULL) + mem_ptr;
    _fmemcpy (dest, buffer, count);
    mem_ptr += count;
    if (mem_ptr > mem_end) mem_end = mem_ptr;
    }
    return (count);
}

static BOOL RewindMem (void)
{
    mem_ptr = 0;
    return (TRUE);
}
```

Extended Memory

XMSFILE.C is the other STORAGE module currently used by DISKDUP. XMSFILE issues device-driver calls to the XMSFILE device built in Part 3.

```
/* xmsfile.c */

/*  This file initialises and co-ordinates modules from the XMSFILE device
    driver in order to supply the DISKDUP utility with extended memory. */

#include    <driver.h>
#include    <smartdrv.h>
#include    <standard.h>

#include    "storage.h"

static DWORD CreateXMS (DWORD);
static BOOL DestroyXMS (void);
static WORD ReadXMS (void _far *, WORD);
static WORD WriteXMS (void _far *, WORD);
static BOOL RewindXMS (void);

struct STORAGE_MEDIUM XMSStorage =
    {0, 0, 0, CreateXMS, DestroyXMS, ReadXMS, WriteXMS, RewindXMS};

/*  The usual access to the XMS server  */

extern void (_far *GetXMSAddress (void)) (void);
void (_far *lpXMS) (void);

/*  XMSFILE is driven through the DOS device driver interface. This just
```

```
    means that information is passed to and fro in a request header. */

/* see seqfile.c for DrvrProc functions */
extern WORD DrvrProcRead (struct REQUEST_HEADER _far *);
extern WORD DrvrProcWrite (struct REQUEST_HEADER _far *);
extern WORD DrvrProcIOCTLWrite (struct REQUEST_HEADER _far *);
extern WORD DrvrProcOpenDevice (struct REQUEST_HEADER _far *);
extern WORD DrvrProcCloseDevice (struct REQUEST_HEADER _far *);

static struct REQUEST_HEADER rqhdr;

/*  Two functions that undercut the device driver interface in XMSFILE but
    which are useful here. */

extern DWORD AdjustRequest (BYTE, DWORD);
extern WORD Destroy (void);

/*  XMSFILE may take extended memory from the SMARTDRV.SYS disk cache. It
    needs some functions from SMARTDRV.C in order to detect SMARTDrive's
    presence and suitability. A flag is left in TRANSFER.C so that the
    XMSFILE device driver code knows to steal memory from the disk cache. */

extern int OpenSMARTDrive (void);
extern BOOL SupportsIOCTL (int);
extern BOOL ReadSMARTDrive (int, struct SD_READ *);
extern BOOL CloseSMARTDrive (int);

extern BOOL bUseSMARTDrive;

static DWORD CreateXMS (DWORD requirement)
{
    DWORD avail;
    WORD status;
    int sdhandle;
    struct SD_READ sd_read;
    WORD largestavailEMBinKB;
    DWORD largestavailEMB;

    /*  Get the XMS entry point and record it for the routines in the
    XMSFILE modules. */

    lpXMS = GetXMSAddress ();
    if (lpXMS == NULL) return (0);

    /*  Find the SMARTDRV.SYS disk cache and check that it uses extended
    memory. */

    sdhandle = OpenSMARTDrive ();
    if (sdhandle != -1) {
    if (SupportsIOCTL (sdhandle)) {
        if (ReadSMARTDrive (sdhandle, &sd_read)) {
        if (sd_read.MemoryType == 1) bUseSMARTDrive = TRUE;
        }
    }
    CloseSMARTDrive (sdhandle);
    }

    /*  Open the XMSFILE "device" then ask it how much of the requirement
    could it meet (in the way of memory it can write to). */

    status = DrvrProcOpenDevice (&rqhdr);
    if (status & D_ERROR) return (0);

    avail = AdjustRequest (D_WRITE, requirement);

    /*  The XMSFILE "device" was given no means to indicate its ability to
    meet part of a request:  its allocations are "all or nothing". If
    it turns out that the XMSFILE algorithm cannot meet the full
    requirement, scale down the request to the size of the largest free
    EMB and try again. This is, of course, a hack that will usually
    handle the situation - but for more sophistication and greater
    coverage, it would be better to improve the routines in XMSFILE. */

    if (avail == 0) {
    _asm {
        mov ah,08h
        call    lpXMS
        mov largestavailEMBinKB,ax
    }
    largestavailEMB = largestavailEMBinKB << 10;
    avail = AdjustRequest (D_WRITE, largestavailEMB);
    }
    return (avail);
}

static BOOL DestroyXMS (void)
{
    if (lpXMS != NULL) Destroy ();
    return (TRUE);
}
```

```
/*  The remaining functions simply phrase the read, write and rewind
    services for DISKDUP storage into terms that XMSFILE understands. */

static WORD ReadXMS (void _far *buffer, WORD count)
{
    rqhdr.lpBuffer = buffer;
    rqhdr.wCount = count;
    DrvrProcRead (&rqhdr);
    return (rqhdr.wCount);
}

static WORD WriteXMS (void _far *buffer, WORD count)
{
    rqhdr.lpBuffer = buffer;
    rqhdr.wCount = count;
    DrvrProcWrite (&rqhdr);
    return (rqhdr.wCount);
}

static BOOL RewindXMS (void)
{
    DWORD temp = 0;
    WORD status;
    rqhdr.lpBuffer = (BYTE _far *) &temp;
    rqhdr.wCount = sizeof (temp);
    status = DrvrProcIOCTLWrite (&rqhdr);
    return (status & D_ERROR ? FALSE : TRUE);
}
```

The Complete Program

Given how many modules it uses (some of which come from other projects in Part 3 of this book), building DISKDUP requires a makefile.

```
!INCLUDE <standard.mak>

SEQFILE_DIR = ..\..\devtools\driver\samples\seqfile
SMARTDRV_DIR = ..\..\xms\smartdrv
XMSFILE_DIR = ..\..\xms\xmsfile

TARGETS : diskdup.exe

mem.obj : mem.c storage.h

storage.obj : storage.c storage.h
```

```
xmsfile.obj : xmsfile.c storage.h

#   SEQFILE.C, SMARTDRV.C and TRANSFER.C are in other directories because
#   they are taken from other projects.

seqfile.obj : $(SEQFILE_DIR)\seqfile.c
  $(COMPILE) $(SEQFILE_DIR)\seqfile.c

smartdrv.obj : $(SMARTDRV_DIR)\smartdrv.c
  $(COMPILE) $(SMARTDRV_DIR)\smartdrv.c

transfer.obj : $(XMSFILE_DIR)\transfer.c
  $(COMPILE) $(XMSFILE_DIR)\transfer.c

#   GETXMS.OBJ and SHIFT.OBJ are in the library directory and are assumed to
#   be current.

diskdup.exe : diskid.obj diskio.obj errormsg.obj gioctl.obj main.obj mem.obj \
              seqfile.obj smartdrv.obj storage.obj transfer.obj xmsfile.obj
  $(LINK) @<<diskdup.lnk
diskid+diskio+errormsg+getxms+gioctl+main+mem+
seqfile+shift+smartdrv+storage+transfer+xmsfile
diskdup.exe;
<<KEEP
```

Readers are invited to improve DISKDUP by removing the requirement that the target disk be already formatted to the same spec as the source disk. Formatting the disk is simple: Before writing each track, you format it—an extra function call to int 21h function 440Dh minor code 42h (Format Track)—and there is hardly more to it.

DoubleSpace

In a lengthy discussion of DOS disk handling, one would think there would be much to say about the DoubleSpace disk compression added in DOS 6. Part 1 of this book showed how DBLSPACE.BIN fits into the DOS startup sequence. However, there isn't much to say about DoubleSpace itself.

As far as this book and its author are concerned, there is nothing fundamental about DoubleSpace. Just like SmartDrive, the DOS task switcher, or Windows, it is an extension that needs to be discussed only in the sense that its presence alters the programming model that DOS programmers must face. This book hasn't discussed the inner workings of such extensions just for their own sake. Except for the unwanted intrusion of the SMARTDRV.EXE int 2Fh API in Part 3, the book is consistent in this.

However, there are certain ways in which DoubleSpace (or any on-the-fly disk compression, such as Stacker, for that matter) *does* change the normal DOS programming model, in other words, ways in which the presence of disk compression simply *can't* be transparent.

For example, if you were attempting to allocate a temporary area (say, one megabyte) of disk space as a DOS file, you would do it in the following steps:

```
create file using appropriate sharing mode (function 6Ch);
advance file pointer by amount required, less one byte (function 42h);
write one byte (function 40h);
rewind file pointer to start of file (function 42h);
```

Writing one byte immediately before the megabyte causes DOS to allocate FAT entries for the desired storage space. In a multi-tasking environment, nobody else can cut in to this storage space unless they go beneath the file system (equivalent to direct access to memory) or they use the same file name (equivalent to a memory handle). This does not mean that the system has said the whole megabyte is writeable, but in the ordinary course of events where there are no physical errors, and the megabyte is secure. In technical terms, physical errors remain possible and the well-written application will be prepared for them, but one thing it will not expect is an error declaring there to be insufficient disk space. Nor will it expect that access might be denied as it tries to write further and further into the file. The application has taken the necessary steps to avoid such nuisances.

In the presence of disk compression software, however, this can no longer be true. Even in the ordinary situation where disk errors do not occur, it is not enough to have DOS allocate storage in the FAT. The amount of available space on the compressed disk depends on *what* you write to the disk. Space could be "reserved" with a one-megabyte run of character 00h, which would compress very nicely and consequently fit onto the disk, yet later it might not have room to hold one megabyte of random numbers, which would not compress at all! It is all too possible that when the application actually comes to write to its temporary file, it will be told that the megabyte is not actually there! The available space depends on *what* you want to write.

This is a small, but intrinsic, incompatibility that arises with disk compression. There is essentially nothing to be done about it. The pity is that it can affect software that embarks on a possibly lengthy operation only after first trying its best to ensure that the necessary disk space will be available.

Index

A20 line, 387-412
  external handlers, 402-412
  HIMEM.SYS handlers, 399-401
  local, global, and physical XMS functions, 388-399
A20 program
  handler.c listing, 408-409
  machine.c listing, 410-411
  makefile listing, 411-412
A20MAP.C listing, 391-396
A20TEST program, 207-209
A20TEST.ASM listing, 208-209
ADDFILES program, 226-239
  ADDFILES.C listing, 226-236
  SFT.C listing, 237-239
ADDFILES.C listing, 226-236
_aNchkstk function, 287
ANSI.SYS, 17-18
API.C listing, 526-528
APPNOTES.TXT file, 8
ARENA.C listing, 183-192
AssignBlock function, 175, 181
Asynchronous drive-swap notification, 636-637
Automatic system configuration, 7-8

Back-filled memory, 40-46
  already outside Windows, 43-46
  memory managers, 41
  Windows enhanced mode, 41-43
_based keyword, 249
BIOS
  describing hardware properties, 7
  drive number, 614-620
  inaccuracies, 8-9
  memory count during initialization, 39-40
  overriding, 8-9
  repair, 59
BIOSDRV.C listing, 614-620
BIOSMEM.C listing, 241
Block device drivers, 9
BLOCKDEV VxD, 57
Boot sectors, 30, 599-608
  accepted hard disk formats, 602-603
  floppy disk drives, 605-606
  floppy disks, 606-607
  hard disk, 601
  OEM name and version, 601-602
  reliance on media descriptor, 607-608
  unrecognized hard disk formats, 603-605
BREAK.C listing, 181
Breakpoints and Windows, 70-72
Buffers
  CONFIG.SYS parsing, 145-150
  kernel, 13
buffers statement, 13
BuildListOfFreeEMBs function, 432
BuildNewSFTs function, 239

C programming language
  functions for device driver commands, 303-304
  in-line XMS function calls, 425-427
  initializing device drivers, 311-331
  resident programming, 249-300
  writing device drivers as programs, 301-347
CHAININT.ASM listing, 286-287
_chain_intr function, 74, 286
chkdsk /f command, 30
Clipboard, 459-478
  initialization, 472-478
  local XMS under enhanced mode, 471-472
  task switching, 466-472
Clock device driver defaults, 9-11
CLOSEALL.C listing, 274-275
CMPXCHG instruction, 371
Code page support, 17-18
COM program memory layout, 205
Command processor, 127-129
  copying to system disk, 31
COMMAND.COM, 10, 23, 127
  critical errors, 676-677
  default error messages, 127
  error code translation, 670-675
  implementing prompt, 11

  - loadhigh command, 23
  - /msg switch, 127, 672
  - /P switch, 129
  - parse errors, 675-676
  - reading command lines, 10
  - shell command, 22
  - standard error messages, 668-670
  - structure on disks, 128
  - support for multiple copies, 128
- comment statement, 145-148
- Compaq and int 13h handling, 56
- COMSPEC system variable, 31
- CON device
  - adding features, 17
  - driver, 7
  - int 16h older functions, 8
- CONFIG.SYS
  - buffers directive, 589
  - changing FILES= statement, 226-239
  - comment statement, 145-149
  - configuration block analysis, 151-154
  - cpsw=on|off statment, 113
  - device statement, 23, 313
  - devicehigh statement, 3, 23, 148
  - dos statement, 22-23
  - dos=high statement, 4
  - dos=umb statement, 3
  - driver and program commands, 149-150
  - drivparm statement /i switch, 54
  - extract, 154
  - extract for typical DOS configuration, 558
  - /f switch, 151
  - general parsing, 147-148
  - install statement, 101
  - /k switch, 150
  - multiple statement passes, 154-155
  - /n switch, 151
  - numlock command, 151
  - parsing buffer, 145-150
  - processing, 145-155
  - switches statement, 149
  - system stabilization, 155
  - tokenism, 146-147
  - user-level system configuration, 8
  - /w switch, 15
- Configuration blocks
  - analysis, 151-154
  - consulting keyboard, 151
  - global properties, 151
  - inclusion, 154
  - menu blocks, 151-152
  - option selection, 152-154
- console, input and special keys, 97-98
- CONSULT.C listing, 528-530
- Conventional memory, 712-714
- copy command, 30, 345
- COUNTRY.SYS, 489
- CPU, Flat Memory Model identification, 370-372
- CRTDRVR program, 242
- CRTDRVR.ASM listing, 317-331
- Ctrl-Alt-Del
  - monitoring keyboard for, 48-53
  - SMARTDRV.EXE and, 52
  - Windows and, 50-52
  - Windows and SMARTDRV.EXE, 53
- Ctrl-P problems, 98-99
- Ctrl-S problems, 99
- Current Directory Structure (CDS), 14, 626

- date command, 22
- DBLSPACE.BIN, 27
  - loading, 28
  - loading early, 156-157
  - modified device driver initialization, 157
- DEBUG, stack sharing scheme, 12
- DescribeCDSProperties function, 633
- Descriptor cache, 356
- Device drivers
  - ANSI.SYS, 17-18
  - C functions for commands, 303-304
  - CON, 7
  - DBLSPACE.BIN, 27
  - default, 7, 32
  - disk block default, 608-620
  - DISPLAY.SYS, 17-18
  - DOS 5 help, 315-316
  - DOS functions initializing, 311-312
  - DRIVER.SYS, 17
  - early execution, 19
  - EGA.SYS, 18
  - EMM386.EXE, 19-20
  - enhancing default support, 17
  - essential components, 301-311
  - example, 332-347
  - extended memory access, 332-333

- file emulation, 333-340
- front end, 306-311
- hardware not known by default, 16-17
- headers, 306-311
- HIMEM.SYS, 19
- hindrance from earlier DOS versions, 316
- initialization in C, 311-331
- initializing default, 138-139
- initializing device, 340
- installable, 16-21
- interfacing with, 20
- loading arbitrary program as, 243-246
- loading resident programs as, 331-332
- managing extended memory block, 340-344
- modified initialization, 157
- MOUSE.COM, 19
- MOUSE.SYS, 19
- NUL, 305
- of convenience, 18-21
- operating modes for devices, 345-347
- PRINTER.SYS, 17
- RAMDRIVE.SYS, 17
- run-time interface library, 312-331
- SETVER.SYS, 19
- SMARTDRV.EXE, 27
- SMARTDRV.SYS, 19
- to file system, 621-633
- transparency, 313
- unknown commands, 304-306
- viewing startup messages, 441
- writing as C programs, 301-347

device keyword, 147, 149
device statement, 18, 23
devicehigh directive, 18
devicehigh keyword, 23, 147-149
- memory fudge, 241-243

devices
- operating modes, 345-347
- parameters, 661

Disk drives and BIOS number, 614-620
DISKCOPY utility, 585
DISKDUP program, 687-719
- /s switch, 694
- DISKID.C listing, 695-700
- DISKIO.C, 700-707
- error codes, 667-685
- MAIN.C listing, 687-694
- makefile listing, 717-718
- MEM.C listing, 712-714
- /SetDrivparmOff switch, 662, 694
- STORAGE.C listing, 708-712
- XMSFILE.C listing, 714-717

DISKHOOK program, 78-91
- manipulating memory allocation, 90
- nesting execution, 93
- SMARTDRV.EXE, 92
- task switcher, 94

DISKHOOK.C listing, 78-90
DISKID.C listing, 695-700
DISKIO.C listing, 700-707
Disks
- absolute read and write, 587-591
- accessing track-by-track, 661-663
- accessing, 700-707
- boot sectors, 599-608
- changes, 60-66
- default block device driver, 608-620
- device driver to file system, 621-633
- device parameters, 659-660
- drive numbering, 598-599
- duplicating, 585
- duplicating through extended memory, 687-719
- enhanced mode and changes, 62-66
- generic IOCTL, 654-666
- get/set access flag, 665
- identifying, 695-700
- logical drive mapping, 633-654
- losing errors, 62
- low level support, 587-666
- management, 585
- media identification, 663-665
- media type, 665
- querying for specific minor codes, 656-659
- simulating change-line errors, 60
- support, 54-56
- temporary storage, 707-712

DISPLAY.SYS, 17-18
DMA errors, correcting at lowest level, 7

DOS
  640K barrier, 8
  booting externally, 140-142
  default regime, 22
  driving Windows, 124-127
  extended partitions, 596-598
  functions, 90-91
  High Memory Area (HMA), 23-24
  memory management, 1-5
  memory problems, 1
  primary partitions, 593-596
  regimes, 22-25
  ROM-executable, 24-25, 106-110
  running as virtual-386 task, 20
  setting version, 19
  Shell, 1
  special keys, 97-99
  system clock, 10-11
  task switching, 1
  versions and IO.SYS, 28-29
DOS 4 and SHARE, 21-22
DOS 5
  accessing extended memory, 4
  device driver help, 315-316
  dos=high, 62
  DOSKEY utility, 11
  EMM386.EXE, 3
  Extended Memory Specification (XMS), 4
  headroom, 134
  High Memory Area (HMA), 4, 15
  loadhigh statement, 3, 42
  losing disk errors, 62
  memory managers, 3
  switches /t command, 9
  SYSINIT module, 47
  system, 132-134
  Upper Memory Block (UMB), 3-4
  WINA20.386 role, 15-16
DOS 6
  HIMEM facility for enumerating EMBs, 438-439
  IO.SYS, 28
  MEM.EXE, 438
dos function, 641
DOS functions
  30h (Get DOS Version), 110, 111, 315
  31h, 312, 314, 316
  3306h (Get DOS Configuration and True Version), 112
  3Fh, 676
  40h, 676
  48h (Allocate Memory), 166
  49h (Free Allocated Memory), 166-167
  4Ah (Set Memory Block Size), 167
  4Bh (EXEC), 111, 193-194, 316
  4Ch, 312
  4Dh (Get Return Code), 313
  59h (Get Extended Error Information), 682-683
  69h (Get/Set Media ID), 664-665
  4400h, 677
  4401h, 677
  4408h (Query Media Removability), 647
  4409h (Get Drive Flags), 645-646
  440Dh (Generic IOCTL for Block Device Drivers), 654-655
  4411h (Query Block Device Driver's IOCTL Support), 656
  4B00h, 313
  4B05h (Set Execution State), 316
  5800h (Get Memory Allocation Strategy), 163-164
  5801h (Set Memory Allocation Strategy), 164
  5801h (Set Upper Memory Link), 165
  5802h (Get Upper Memory Link), 165
  5D0Ah (Set Extended Error Information), 684
  6F00h (Get Starting Segment for ROM Programs), 109
  6F01h (Set Starting Segment for ROM Programs), 109
  6F02h (Get Address of ROM Exclusion List), 109, 110
  6F03h (Set Address of ROM Exclusion List), 110
  error codes returned in ax, 681
  initializing device drivers, 311-312
_dos_keep function, 276, 313-314, 316

DOS kernel, 104-127. *See also* Kernel
  nondestructive read command to device driver, 10
dos statement, 22-23
dos=umb statement, 3
DOSINHMA.C listing, 102-103
DOSKEY utility, 10-11
DOSLEVEL.C listing, 627-632
DOSMGR Device Broadcast
  00H (Establish DOSMGR Support), 118
  01H (Enable Specialized Windows Support), 119
  02H (Disable Specialized Windows Support), 120
  03H (Get System Structure Sizes), 121
  04H (Get Instancing Exemptions), 121-122
  05H (Get Size of Device Driver), 122
DOSMGR V86 API functions
  0000h (Get DOSMGR Version), 125
  0001h (Set Critical Focus), 125
  0003h (Enter Critical Section), 126
  0004h (Get VM ID Byte), 126
  0005h (Inform Windows of Possible Media Change), 126-127
DOSMGR virtual device driver, 14
_dospawn function, 209-210
DOSPAWN.C listing, 210-220
DOSSEG comment, 249
DOSSHELL.COM, 22
_dos_special_allocmem function, 169, 175
DOSSWAP task switcher, 541, 547-548
  fragmented extended memory, 573-577
_dos_write function, 346
DOSX.EXE, 560-561
  CPU identification, 370
DoubleSpace, 718-719
Drive Parameter Block (DPB), 621-625
DRIVER program, 242
DRIVER.ASM listing, 306-311
DRIVER.SYS, 17
  /d switch, 633
Drives
  as logical entities, 14
  asynchronous swap notification, 636-637
  coordinating directory trees across, 625-633
  Current Directory Structure (CDS), 14
  logical mapping, 633-654
  number attached, 13-14
  Windows inactive shared, 641
drivparm statement, 9, 54
DRVINFO program
  MAIN.C listing, 642-645
  makefile listing, 652-653
DRVINFO.C listing, 648-651
DRVMAP.C listing, 638-641
DrvrProcBadCommand function, 304
DrvrProcInit function, 313-314
DSWAP task switcher, 541, 547-548
EATBLOCK program, 168-183
  BREAK.C listing, 181
  code organization, 168-169
  EATBLOCK.C listing, 175-180
  MALLOC.C listing, 169-171
  MEMBLOCK.C listing, 171-175
  READHEX.C listing, 180
  source code files, 169
EATBLOCK.C listing, 175-180
EDLIN, 62-63
EGA Register Interface, 18
EGA.SYS, 18
EMM386.EXE, 3, 19-20
  deactivating, 36
  enhanced mode and, 43-44
  int 19h and, 36-37
  noems keyword, 558
  not restoring interrupt vectors, 36
_enable function, 289
Enhanced mode, 469-470
  back-filled memory, 41-43
  buffers system, 63
  Ctrl-Alt-Del and, 50-52
  disk changes and, 62-66
  EMM386 and, 43-44
  int 19h and, 37
  local XMS clipboard under, 471-472
  virtual machines (VMs), 63
  XMSWATCH program, 582-584

Error codes, 667-685
  COMMAND.COM translation, 670-675
  critical errors, 676-677
  extended information, 680-685
  network errors, 677-680
  parse errors, 675-676
  standard messages, 668-670
  translation, 667-680
ERRORMSG.C listing, 673-675
EXE programs
  memory layout, 201-203
  overlay relocation, 203-205
exit function, 313
ExitKernel function, 38
ExitWindows function, 38
ExitWindowsExec function, 38
Expanded Memory Managers, 2
Expanded Memory Specification (EMS), 2
Extended memory, 714-717
  80286 LOADALL instruction, 373-379
  80386 flat memory model, 357-372
  80386 LOADALL instruction, 379-385
  A20 line, 387-412
  blocks, 427-437
  cooperation, 416-418
  descriptor cache, 356
  device driver access, 332-333
  device driver block management, 340-344
  disbursing, 423-439
  DOS 5 access, 4
  DOS 6 HIMEM facility for enumerating blocks, 438-439
  High Memory Area (HMA), 423-425
  implementation in HIMEM.SYS, 442-456
  in-line function calls from C, 425-427
  local, global, and physical XMS functions, 388-399
  management, 349-354
  memory above 64MB, 420-421
  noncontiguous, 418-419
  outline, 349-354
  real mode access, 355-385
  shadow RAM, 419-420
  sources, 413-421
  VDISK, 414-416
Extended memory blocks
  HIMEM.SYS flaws with reallocation, 577-582
Extended Memory Specification (XMS), 4, 349-354
  accessing disks, 700-707
  clipboard, 459-478
  disk identification, 695-700
  duplicating disks, 687-719
  environments, 557-584
  extending, 505-549
  front end for XMS extension, 513-519
  functions, 351-352
  global allocation from task switcher, 470-471
  global and local handles, 539
  handler chain arrangement, 506
  hookability, 505-512
  memory from SMARTDrive, 478-503
  modification by task switchers, 538-549
  monitor, 551-584
  non-trivial handling in standard mode, 565-567
  spying on handle allocations, 519-549
  temporary disk storage, 707-712
  tracing the chain, 510-512
  usage, 459-504
EXTERN directive, 304-306
External driver functions
  00h (Availability Check), 612
  02h (Pass Call from External Driver), 613
  03h (Get Address of First Unit Table), 613

Far pointers, preventing interruption while changing public, 73-75
_fastcall attribute, 390-391
_fastcall keyword, 425
FastDisk, 57
  int 2Fh function 13h requirements, 75-76
  proceeding with installation, 76-78
FASTOPEN program, 490
fcbs command, 13
File Control Block (FCB), 13
  incompatibility opening files on hard disk, 21-22
  obsolescence, 14
File emulation and device drivers, 333-340

- File system, 621-633
  - coordinating directory trees across all drives, 625-633
  - Drive Parameter Block (DPB), 621-625
  - enabling, 143-145
- Files
  - deleting, finding, and renaming, 14
  - executable formats, 197-198
- files command, 13
- FILES= statement, changing, 226-239
- Flat Memory Model, 357-372
  - changing segment limit, 362-369
  - CPU identification, 370-372
  - real mode instability, 372
  - segment limit, 357-362
  - virtual machine detection, 369-370
- Floating-point emulation, 294-295
- Floppy disk drives
  - boot sectors, 605-606
  - standard controllers, 9
- Floppy disks
  - boot sectors, 606-607
- FollowXMSChain function, 509-510
- FP.C listing, 293-294
- FPFIXUPS.ASM listing, 294-295
- FREE.C listing, 432-437
- Functions, 90-91
  - checking for Ctrl-Break, 12
- GetArenaSegment function, 183
- GetBIOSDrive function, 620
- GetBIOSInt13h function, 92
- getch function, 97-98
- GetDiskCharacteristics function, 687, 695
- GetDiskMediaID function, 687, 695
- GetDriveType function, 634, 641
- GetErrorMsg function, 675
- GETINT13 listing, 67-69
- GETLIMIT.ASM listing, 358-362
- GetSFTSize function, 236
- GetSubArenaSegment function, 183
- GETXMS.C listing, 350-351
- GIOCTL.C listing, 657-659
- GlobalDisable function, 390
- GlobalEnable function, 389
- GRAPHICS utility, 35

- HANDLER.C listing, 295-296, 408-409
- HANDLERS.C listing, 510-511
- Hard disks
  - accepted boot sector formats, 602-603
  - boot sectors, 601
  - controllers, 9
  - drive numbering, 598-599
  - extended DOS partitions, 596-598
  - incompatibility opening files, 21-22
  - int 2Fh interface, 19
  - IOCTL interface, 19
  - large, 59
  - partition table, 592-593
  - partitioning, 591-598
  - primary DOS partitions, 593-596
  - reading to or writing from, 19
  - unrecognized boot sector formats, 603-605
- _harderr function, 200
- _hardretn function, 200
- Hardware
  - describing properties, 7
  - enhancing default support, 17
  - nonstandard, 27
  - not known by default, 16-17
- HasDiskChanged function, 694-695
- High Memory Area (HMA)
  - 8086, 4
  - buffers system, 23
  - DOS, 23-24
  - DOS 5, 4, 15
  - managing, 23
  - measuring DOS usage, 102-104
  - paragraph boundaries, 4
  - reporting on, 423-425
  - VDISK headers, 46-47
- HIMEM functions
  - 00h (Get XMS version), 444-445
  - 01h (Allocate HMA), 445
  - 02h (Free HMA), 445-446
  - 03h (Global Enable A20 Line), 446
  - 04h (Global Disable A20 Line), 446

    05h (Local Enable A20 Line), 447
    06h (Local Disable A20 Line), 447
    07h (Query A20 Line Status), 448
    08h (Query Free Extended Memory), 448
    09h (Allocate Extended Memory Block), 449
    0Ah (Free Extended Memory Block), 450
    0Bh (Move Data to or from Extended Memory Block), 451
    0Ch (Lock Extended Memory Block), 452
    0Dh (Unlock Extended Memory Block), 453
    0Eh (Get Information about Extended Memory Block), 454
    0Fh (Reallocate Extended Memory Block), 454-455
    88h (Query Free Extended Memory), 448
    89h (Allocate Extended Memory Block), 449
    8Eh (Get Information about Extended Memory Block), 454
    8Fh (Reallocate Extended Memory Block), 454-455
HIMEM.SYS, 19, 23, 441-458
  /a20control:off switch, 399, 447
  A20 line handlers, 399-401
  command line configuration, 441-442
  /cpuclock:on switch, 401
  delayed hook, 417-418
  /eisa switch, 421
  /hmamin switch, 423
  /int15 switch, 300, 416-418, 444
  lack of Upper Memory Block (UMB) management, 4
  /m:at switch, 401
  /machine switch, 399, 401
  /no switch, 419
  /numhandles switch, 437
  reallocate EMB function, 456-458
  serious flaws with extended memory block reallocation, 577-582
  /shadowram:off switch, 420
  switches, 441-442
  /u switch, 419
  unacceptability of version 3.xx, 456-458
  usage, 4-5
  /verbose switch, 441
  version numbers, 445
  /z switch, 441
  XMS implimentation, 442-456
  XMS routines, 19
hma.c listing, 423-425

Idle loops, 10-11
ifs keyword, 147, 149
IFSFUNC program, 500
include statements, 150, 154
InitSMARTDrive routine, 500
inp function, 289
Input handles, checking for Ctrl-C, Ctrl-P and Ctrl-S, 12
InquireSystem function, 641
install directive, 21, 312
install keyword, 147, 149
INSTALL.C listing, 523-526
Install_V86_Break_Point function, 70, 538
Instance data, 97-102
  subletting HMA, 100-102
  Windows/386, 123
INSTANCE.386 file, 123
int 00h, 34
int 01h, 33-34
int 03h, 33-34
int 04h, 33-34
int 05h, 35
int 06h (Invalid Opcode), 358
int 09h, 50, 52
int 0Dh (GP Fault), 358
int 10h, 18, 33
int 11h, 599
int 12h function 1605h, 471
int 13h, 7, 9, 19, 31, 54-56, 76, 590, 592-593, 595-596, 598, 601, 614, 620
  change-line errors, 55
  DMA transfers, 54, 55
  ECC data corrections as errors, 55
  machine-specific handling, 56
  managing, 57-58
  PS/2, 54
int 13h function 01h, 54
int 13h function 08h (Get Drive Parameters), 54, 599, 602-603, 605-606
int 13h function 15h, 54
int 13h function 20h, 665-666
int 14h, 35
int 15h (Monitor for Ctrl-Alt-Del), 33, 48-53, 332
  carry flag, 49

- int 15h function 4Fh, 49-50, 52
- int 15h function 87h (Move Extended Memory Block), 47, 56, 332-333, 345, 355, 388-389, 452
- int 15h function 88h (Get Extended Memory Size), 340, 413-414, 419-421, 444, 459
  - hooking, 416
- int 15h function 89h (Enter Protected Mode), 388
- int 15h function D800h (EISA Read Slot Configuration Information), 420
- int 15h function D801h (EISA Read Function Configuration Information), 420
- int 16h, 11
  - CON device driver, 8
- int 16h, function 10h, 7
- int 17h, 35
- int 19h (Reboot), 24, 33-47, 415
  - additional changes, 39-41
  - defeated by memory managers, 36
  - EMM386 and, 36-37
  - enhanced mode and, 37
  - exiting Windows, 38
  - interrupts exempted from precautions, 34-35
  - IO.SYS handler, 47
  - protection, 37
  - VDISK headers and High Memory Area (HMA), 46-47
- int 1Bh, 33
- int 1Ch (Timer), 36
- int 21h, 7, 23, 199-200, 311-312, 316, 587
  - duplication of functions, 491
  - extra functions, 24
  - FCB functions, 14
- int 21h function 08h (Read Keyboard Without Echo), 10
- int 21h function 0Ah (Buffered Keyboard Input), 10-11
- int 21h function 11h (FCB Find First), 664
- int 21h function 25h, 505
- int 21h function 2Ch (Get Time), 11
- int 21h function 32h, 621, 627
- int 21h function 33h, 113
- int 21h function 34h (InDOS Flag), 501
- int 21h function 38h (Get/Set Country Information), 489-490
- int 21h function 3Fh, 10
- int 21h function 4Bh (EXEC), 193-196, 202-204
- int 21h function 52h (Get SysVars), 143, 500-501, 626
- int 21h function 53h, 624
- int 21h function 58h, 3
- int 21h function 60h (TRUENAME), 633
- int 21h function 64h, 10
- int 21h function 65h (Get Extended Country Information), 489-490
- int 21h function 66h (Get/Set Global Code Page), 489-490
- int 21h function 68h, 487
- int 21h function 69h, 663
- int 21h function 6Dh (ROM Find First), 107-108
- int 21h function 6Eh (ROM Find Next), 107-109
- int 21h function 0601h, 654
- int 21h function 0802h, 613
- int 21h function 0803h (Get Address of First Unit Table), 614
- int 21h function 1206h, 677
- int 21h function 2522h, 50
- int 21h function 3303h, 113
- int 21h function 3304h, 113
- int 21h function 3305h (Get Startup Drive), 599
- int 21h function 3306h, 316
- int 21h function 4400h (Get Device Data), 346
- int 21h function 4401h (Set Device Data), 346
- int 21h function 4402h (IOCTL Read), 78, 346, 479
  - data structure, 480-481
- int 21h function 4403h (IOCTL Write), 346, 481-482
- int 21h function 4409h, 646
  - minor code 4Ch (Start Code Page Prepare), 18
- int 21h function 440Dh, 591
  - minor code 40h, 54
  - minor code 42h (Format Track), 718
- int 21h function 440Eh (Get Logical Drive Map), 634-635
- int 21h function 440Fh (Set Logical Drive Map), 635-636
- int 21h function 4B01, 210
- int 21h function 4B03h, 201
- int 21h function 4B05h (Set Execution State), 205-209

int 21h function 5D06h, 196, 501
int 21h function 5F02h, 646
int 21h function 5F06h, 626
int 24h, 316, 676
  exceptions, 61-62, 199-200, 306
int 25h (Absolute Disk Read), 587-590
int 26h (Absolute Disk Write), 587-590
int 28h, 10-11
int 29h, 33
int 2Ah, function 84A, 10
int 2Fh, 19, 57-58, 671
  accessing IO.SYS Unit Tables, 612-614
  support for booting DOS externally, 142
  support for multiple copies of COMMAND.COM, 128
  XMS support, 349-351
int 2Fh function 05h, 678-679
int 2Fh function 08h, 57
int 2Fh function 11h, 678
int 2Fh function 12h, 490, 573
int 2Fh function 13h, 57-60
  applications, 59-60, 66-78
  availability, 72
  backword compatibility, 78
  behind the interface, 66-69
  BIOS repair, 59
  FastDisk requirements, 75-76
  finding ROM, 76
  forward compatibility, 76-78
  installing int 13 handler, 78-94
  interrupt windows, 72-73
  large hard disks, 59
  large sectors, 59-60
  preventing interruption while changing public far pointers, 73-75
  safety and compatibility, 94-95
  simulating disk change-line errors, 60
int 2Fh function 16h, 96
int 2Fh function 0801h, 609
int 2Fh function 0803h, 609
int 2Fh function 112Bh, 655
int 2Fh function 1206h, 682
int 2Fh function 1216h (Get SFT Entry), 236
int 2Fh function 122Eh, 671-672
int 2Fh function 1603h (Get Windows/386 Instance Data), 123
Int 2Fh Function 1605h (Windows Startup Broadcast), 15, 36, 96, 98, 116, 483, 562
Int 2Fh Function 1606h (Windows Exit Broadcast), 51, 96, 117, 483
int 2Fh function 1607h (Device Intialization Broadcast), 76, 100
int 2Fh function 1680h, 11
int 2Fh function 1684h (Locating Virtual Device API), 96, 124, 636
int 2Fh function 4300h (XMS Availability Check), 350
int 2Fh function 4309h, 438-439
int 2Fh function 4310h (Get XMS Calling Address), 350, 390, 425, 540
int 2Fh function 4601h, 122
int 2Fh function 4602h, 123
int 2Fh function 4810 (Read Command Line), 11
int 2Fh function 4A00h (Broadcast Preceding Logical Floppy Drive Swaps), 637
int 2Fh function 4A01h (Query HMA Space), 101, 137
int 2Fh function 4A02h (Allocate HMA Space), 101, 137
int 2Fh function 4A10h, 50, 483
int 2Fh function 4B02h, 548
int 2Fh function 4B03h, 137
int 2Fh function 4B05h, 98, 137
int 2Fh function 4B20, 38
int 4Bh, 36
int 67h, 36
int 80h, 36
INT13 virtual device, 57
INT15MEM program, 295-300
  extended memory block management, 340-344
  HANDLER.C listing, 295-296
  initializing device, 340
  loading as device driver, 331-332
  MAIN.C listing, 296-299
  makefile for device driver version of INT15MEM.EXE, 344

    MAKEFILE.C listing, 300
  _interrupt function, 296
  _interrupt keyword, 301
Interrupts
  exempted from int 19h precautions, 3435
  resident program control, 285-287
IO.SYS, 7, 23, 27-30
  A20 control, 32-33
  apportioning HMA allocation, 101
  automatic system configuration, 7-8
  configurable system support, 11
  copying to system disk, 31
  DOS 6, 28
  DOS versions and, 28-29
  drive tables, 609-614
  error conditions in critical sections, 96
  general system support, 7
  high and low segments, 31-33
  int 19h handler, 47
  interrupt handlers, 33
  loader, 29-30
  modularity, 28
  multiplex interfaces, 57-58
  restoring vectors, 34
  stacks command, 11
  SYSINIT module, 28-29
  user-level system configuration, 8
  Windows support, 96
IOCTL interface, 19
is80386.c listing, 371-372
ISFUNC.EXE, 678-679
join command, 14
JOIN utility, 625

KEEPC.H listing, 289-290
KEEPEXIT.ASM listing, 276-283
KEEPINIT.ASM listing, 255-258
KEEPINIT.INC listing, 253-255
KEEPSEGS.ASM listing, 292
KEEPSEGS.INC listing, 250-252
Kernel, 104-122
  activating, 135-145
  booting DOS externally, 140-142
  buffers, 13
  calling, 137-138
  configuring at user-level, 12-15
  direct access to data, 500-502
  enabling file system, 143, 145
  finite resources, 13-14
  initializing default device drivers, 138-139
  interrupt services for DOS services, 139
  marking progress, 137
  memory and process managment, 140
  moving data to low memory, 136
  number of drives, 13-14
  OEM numbers, 111-113
  planning ahead for moving code, 136-137
  setting break=on, 12
  sidedoor entrance, 489-493
  signaling critical multitasking software sections, 10
  System File Table (SFT), 13
  system-wide A20 implications, 15-16
  SYSVARS structure, 142-143
  variables, 113-115
  variations, 110-113
  Windows 3.0 real mode, 123
  Windows support, 115-122
KillParent function, 226

lastdrive command, 14
LIM EMS 3.2, 2
LINK command, 331
LIST.C listing, 429-431
listings. *See* program listings
LOADALL instruction
  80286, 373-379
  80386, 379-385
  descriptor cache format for structure, 378-379, 381
  machine state structure, 377-380
LOADFIX utility, 15
loadhigh command, 3, 23
_loadds keyword, 302
LocalDisable function, 389
LocalDisableA20 function, 391
LocalEnable function, 388-390
LocalEnableA20 function, 390-391
Logical drives
  building map, 638-641
  descriptions, 645-654

- mapping, 633-654
- scanning available, 641-645

MACHINE.C listing, 410-411
main function, 21, 313
MAIN.C (DISKDUP) listing, 687-694
MAIN.C (DRVINFO) listing, 642-645
MAIN.C (XMSFILE) listing, 472-477
MAIN.C (XMSWATCH) listing, 551-555
MAIN.C listing, 296-299, 366-369, 404-407
MAKEFILE (DISKDUP) listing, 717-718
MAKEFILE (DRVINFO) listing, 652-653
MAKEFILE for device driver version of INT15MEM.EXE listing, 344
MAKEFILE listing, 290-291, 411-412, 503-504
MAKEFILE.C (XMSWATCH) listing, 555-556
MAKEFILE.C listing, 300
MALLOC.C listing, 169-171
MaxBPs keyword, 71
Media Check command, 65
mem /d command, 46
MEM program, 504
- /d switch, 564

MEM.C listing, 712-714
MEM.EXE, 438
MEMBLOCK.C listing, 171-175
nemcpy function, 289
Memory
- 640K barrier, 1-2
- 8086 usage, 1
- 80286 usage, 2
- 80386 usage, 2
- accessing upper, 2-3
- back-filled, 40-46
- bank-switching, 2
- block selection by address, 168-183
- kernel management, 140
- layout for COM programs, 205-209
- layout for EXE programs, 201-203
- management, 1-5
- page access, 2
- problems, 1-3

Memory allocation, 163-192
- arena chain, 183-192
- strategy and upper memory linkage, 163-168

Memory managers
- back-filled memory, 41
- defeating int 19h, 36
- EMM386.EXE, 3
- QEMM386.SYS, 44

MEMOWNER.C listing, 511-512
MEMSTRAT.C listing, 90-91
Menu blocks, 151-152
menucolor keyword, 151
menudefault keyword, 150-152
menufault statement, 153
menuitem keyword, 150-154
MODE, 35
mode con cp refresh, 18
mode con cp select=nnn, 18
MONITOR.C listing, 520-523
MOUSE.COM, 19, 37
MOUSE.SYS, 19, 37
MSCDEX.EXE, 679-680
- direct access to kernel data, 501

MSDOS.SYS, 8, 23, 27, 30, 104-127. *See also* Kernel
- copyig to system disk, 31
- high and low segments, 105-106
- ROM-executable DOS, 106-110

NLSFUNC utility, 489-490
NoLow64KPaging=true statement, 16
Noncontiguous extended memory, 418-419
NUL device driver, 305
numlock keyword, 151-152
NEXIT.ASM listing, 284-285

Open function, 467

PAGEFILE device, 57
Partitions, 591-598
- extended DOS, 596-598
- partition table, 592-593
- primary DOS, 593-596

PC-AT, int 13h handling, 56
PRINT utility, 488-489
PRINTER.SYS, 17
printf function, 300
Printing, redirecting from parallel to serial ports, 35
Processes, kernel management, 140
PROGNAME.ASM listing, 262-264
Program execution, 193-239
- changing FILES= statement in CONFIG.SYS, 226-239
- file formats, 197-198
- preparing for DOS exceptions, 199-205

- program environment, 196-197
- SPOTLOAD program, 209-225
- state, 205-209

Program listings
- A20MAP.C, 391-396
- A20TEST.ASM, 208-209
- ADDFILES.C, 226-236
- API.C, 526-528
- ARENA.C, 183-192
- BIOSDRV.C, 614-620
- BIOSMEM.C, 241
- BREAK.C, 181
- CHAININT.ASM, 286-287
- CLOSEALL.C, 274-275
- CONSULT.C, 528-530
- CRTDRVR.ASM, 317-331
- DISKHOOK.C, 78-90
- DISKID.C, 695-700
- DISKIO.C, 700-707
- DOSINHMA.C, 102-103
- DOSLEVEL.C, 627-632
- DOSPAWN.C, 210-220
- DRIVER.ASM, 306-311
- DRVINFO.C, 648-651
- DRVMAP.C, 638-641
- EATBLOCK.C, 175-180
- ERRORMSG.C, 673-675
- FP.C, 293-294
- FPFIXUPS.ASM, 294-295
- FREE.C, 432-437
- GETINT13, 67-69
- GETLIMIT.ASM, 358-362
- GETXMS.C, 350-351
- GIOCTL.C, 657-659
- HANDLER.C, 295-296, 408-409
- HANDLERS.C, 510-511
- HMA.C, 423-425
- INSTALL.C, 523-526
- is80386.c, 371-372
- KEEPC.H, 289-290
- KEEPEXIT.ASM, 276-283
- KEEPINIT.ASM, 255-258
- KEEPINIT.INC, 253-255
- KEEPSEGS.ASM, 292
- KEEPSEGS.INC, 250-252
- LIST.C, 429-431
- MACHINE.C, 410-411
- MAIN.C, 296-299, 366-369, 404-407
- MAIN.C (DISKDUP), 687-694
- MAIN.C (DRVINFO), 642-645
- MAIN.C (XMSFILE), 472-477
- MAIN.C (XMSWATCH), 551-555
- MAKEFILE, 290-291, 411-412, 503-504
- MAKEFILE (DISKDUP), 717-718
- MAKEFILE (DRVINFO), 652-653
- MAKEFILE for device driver version of INT15MEM.EXE, 344
- MAKEFILE.C, 300
- MAKEFILE.C (XMSWATCH), 555-556
- MALLOC.C, 169-171
- MEM.C, 712-714
- MEMBLOCK.C, 171-175
- MEMOWNER.C, 511-512
- MEMSTRAT.C, 90-91
- MONITOR.C, 520-523
- ONEXIT.ASM, 284-285
- PROGNAME.ASM, 262-264
- RDWRID.ASM, 397
- READBOOL.C, 225
- READHEX.C, 180
- SDCONFIG.C, 567-573
- SDINIT.C, 502-503
- SEGDEBUG.ASM, 271-273
- SEQFILE.C, 334-340
- SETLIMIT.ASM, 363-384
- SFT.C, 237-239
- SHIFT.C, 288
- SMARTDRV.C, 491-499
- SPAWN.C, 243-244
- SPOTLOAD.C, 220-225
- STORAGE.C, 708-712
- SWAPREFS.ASM, 265-270
- SWAPSEGS.ASM, 259-262
- TEST.C, 305
- TRANSFER.C, 340-344
- TRANSFER.C (XMS version), 460-466
- VERSION.C, 353
- VM.C, 370
- WATCHED.C, 530-538
- XA20.C, 403-404
- XMS.ASM, 513-519
- XMSCHAIN.C, 507-509

  - XMSFILE.C, 714-717
  - XMSFUNC.C, 426-427
  - XMSFUNC.H, 425-426
  - XMSWATCH.H, 427-429
- Programs
  - .EXE header, 3
  - environment, 196-197
  - headers in .EXE format, 198
  - in ROM, 107-110
  - INT15MEM, 295-300
  - limiting initial memory requirement, 3
  - loader, 209-225
  - memory location assumptions, 3
  - resident, 246-300
  - running, 21-22
  - task-switching, 1
  - XMSMMGR.EXE, 19
- ROTOMAN.SYS, 134
- S/2 and int 13h, 54

- EMM386.SYS, 44
  - nofill keyword, 41
  - Stealth feature enabled, 78
  - Windows 3.0 standard mode and, 562-565

- AM, from address zero upwards, 39-40
- AMDRIVE.SYS, 17
- DWRID.ASM listing, 397
- EADBOOL.C listing, 225
- adDisk function, 687, 700
- EADHEX.C listing, 180
- adStorage function, 707
- adTrack function, 694, 700
- al mode, accessing extended memory, 355-385
- REBOOT VxD, 51
- REDIR.EXE, 678-680
- References, swapping, 262-270
- rem keyword, 151
- Removable media, 61
- REP INSB instruction, 16
- ReportFreeEMBs function, 432
- _RESIDENT keyword, 296
- Resident programs, 246-300
  - avoiding compiler library calls, 287-289
  - building libraries, 289-291
  - debugging, 270-273
  - importing material for residence, 292-295
  - interrupt control, 285-287
  - loading, 21
  - loading as device driver, 331-332
  - preserving extended error information, 684-685
  - rearrangement, 252-258
  - releasing system resources, 273-275
  - segmentation, 250-252
  - support, 273-275
  - swapping references, 262-270
  - swapping segments, 258-262
  - terminating and staying resident, 275-285
  - writing, 295-300
- RestoreKernelData function, 500
- ROM
  - finding, 76
  - functions, 108-110
- ROM-executable DOS, 24-25, 106-110
  - programs in ROM, 107-110
- run-time interface library device drivers, 312-331
- run-time library device drivers, 312-315
- RunTestShell function, 175

- Safety and int 2Fh function 13, 94-95
- SaveKernelData function, 500
- _saveregs key word, 302
- SDCONFIG.C listing, 567-573
- SDINIT.C listing, 502-503
- Sectors, large, 59-60
- SEGDEBUG.ASM listing, 271-273
- SEGLIMIT program, 372
  - MAIN.C listing, 366-369
  - SETLIMIT.ASM listing, 363-365
- Segments
  - marking as resident, 250-252
  - swapping, 258-262
- SEQFILE.C listing, 334-340
- set command, 197
- SetDrivparm function, 700
- SETLIMIT.ASM listing, 363-365, 381-384
- SETVER.EXE, 19
- SFT.C listing, 237-239
- Shadow RAM, 419-420
- SHARE utility, 490
- Shell, 312
  - guarding against termination, 22
  - running, 196-197
  - task switcher XMS modification, 538-549
- shell keyword, 22, 147, 149
- SHIFT.C listing, 288

SMARTDrive
  DOS reentrancy, 488-89
  write-behind cache, 49-50
 XMS memory from, 478-503
SMARTDRV functions
  00h (Installation Check and General Query), 483
  01h (Commit Cache), 483
  02h (Restart Cache), 484
  03h (Get/Set Cache Status), 484
  04h (Get Cache Size), 485
  05h (Get Double Buffering Status), 485
  06h (Check Drive Cacheability), 486
  07h (Get Original Driver Address), 486
  0Ah (Get Element Status Table), 487
  1234h (Display Serious Error Message), 487
SMARTDRV.C listing, 491-499
SMARTDRV.EXE, 27, 37, 478-479
  Ctrl-Alt-Del and, 52
  /d+ switch, 485
  disk changes, 61-62
  DISKHOOK program, 92
  interface, 482-503
  /n switch, 487
  /r switch, 487
  Windows and Ctrl-Alt-Del, 53
  /y switch, 487
SMARTDRV.SYS, 19
  /c switch, 49
  configuring, 567-573
  /d switch, 50
  disk cache, 478
 interface, 479-482
SPAWN.C listing, 243-244
spawnvp function, 209-210
Special keys, 97-100
SPOTLOAD program, 209-225
  DOSPAWN.C listing, 210-220
  READBOOL.C listing, 225
  SPOTLOAD.C listing, 220-225
SPOTLOAD.C listing, 220-225
Stack
  configuring pool for hardware interrupts, 37
  effect of hardware interrupts, 12
  size problems, 11
  sharing scheme, 12
  system overloads, 96
stacks command, 11
Standard mode
  non-trivial XMS handling, 565-567
  QEMM386.SYS and, 562-565
  task switcher XMS modification, 538-549
  WINOA286.MOD, 540
  XMSWATCH program, 559-577
Startup sequence, 131-157
  activating DOS kernel, 135-145
  DOS 5, 132-134
  juggling the sequence, 131-134
  loading DBLSPACE.BIN early, 156-157
  processing CONFIG.SYS, 145-155
  system before DOS 5, 131-132
STORAGE.C listing, 708-712
strlen function, 289
submenu keyword, 150-154
SUBST utility, 625
SWAPREFS.ASM listing, 265-270
SWAPSEGS.ASM listing, 259-262
switches /k statement, 8, 150
switches /n statement, 153
switches /t statement, 9
switches keyword, 147, 149
SYMDEB, stack sharing scheme, 12
SYS utility problems, 30-31
SYSINIT, parameters to initialize MSDOS.SYS, 135
SysInitVars, 113
System
  before DOS 5, 131-132
  DOS 5, 132-134
  juggling, 131-134
System clock, 10-11
  default device driver, 10-11
  midnight flag, 9-11
System configuration, 7-25
  automatic, 7-8
  four DOS regimes, 22-25
  running programs, 21-22
  user-level, 8-21
System disk, copying command processor to, 31
System File Table (SFT), 13
System files, unable to transfer to floppy disk, 30-31

- System footprint, 27-129
  - command processor, 127-129
  - disk support, 54-56
  - high and low segments, 31-33
  - IO.SYS, 27-30
  - monitoring keyboard for Ctrl-Alt-Del, 48-53
  - MSDOS.SYS, 104-127
  - preparing for warm boot, 33-47
  - Windows support, 96-104
- System variables, 113-115
- SYSTEM.INI
  - KybdReboot=Off, 51-52
  - KybdReboot=on, 53
  - Local=CON statement, 100
  - LocalReboot=Off, 51, 53
  - NoEMMDriver=on statement, 42
  - PerformBackfill statement, 42
  - SystemROMBreakPoint setting, 69
  - [386Enh] section, 15-16, 77
- SYSVARS structure, 113-115, 142-143

- Task switchers, 1, 60, 466-472
  - DISKHOOK program, 94
  - DOSSWAP, 541
  - DSWAP, 541
  - global XMS allocation from within, 470-471
  - localized extended memory, 467-468
  - task separation, 468-470
  - Windows enhanced mode, 469-470
  - WSWAP, 541
  - XMS modifications, 538-549
- TEST.C listing, 305
- ThreadXMSChain function, 509, 523-526
- time command, 22
- TRANSFER.C (XMS version) listing, 460-466
- TRANSFER.C listing, 340-344
- TSRs (terminate-and-stay-resident programs). *See* Resident programs

- Upper memory, accessing, 2-3
- Upper Memory Block (UMB)
  - DOS 5, 3-4
  - HIMEM.SYS lack of management, 4
  - problems loading programs into, 3-4
- UseInt13Trap function, 614
- UseIOSYSDriveTables function, 614
- User-level system configuration, 8-21
  - CONFIG.SYS, 8
  - configurable system support, 11
  - configuring kernel, 12-15
  - installable device drivers, 16-21
  - IO.SYS, 8
  - overriding BIOS, 8-9
  - reverting to old behavior, 9-10

- VDISK driver
  - extended memory, 414-416
  - headers and Ctrl-Alt-Del, 48
  - headers and High Memory Area (HMA), 46-47
- ver /r command, 113
- VERSION.C listing, 353
- Video controller registers, 18
- Virtual machines (VMs)
  - enhanced mode, 63
  - Flat Memory Model detection, 369-370
- Virtual Memory Manager (VMM)
  - _Allocate_Global_V86_Data_Area service, 16
  - Release_Time_Slice service, 10
- VM.C listing, 370
- Volume labels, 14
  - management, 65
- VSAFE, 50

- Warm boot, preparing for, 33-47
- WATCHED.C listing, 530-538
- WDCTRL virtual device, 57
  - upper memory as ROM, 90
  - Windows startup, 92
- WDCTRLAltStatus0 keyword, 77
- WDCTRLAltStatus1 keyword, 77
- WIN.COM, 22, 38, 197
- WIN386.EXE, CPU identification, 370-371
- WINA20.386, 15
  - installing, 15
  - role in DOS 5, 15-16
- windir environment variable, 197

Windows
  breakpoints, 70-72
  Ctrl-Alt-Del and, 50-52
  driving from DOS, 124-127
  enhanced mode, 469-470
  exiting and int 19h, 38
  inactive shared drives, 641
  instance data, 97-102
  kernel support, 115-122
  SMARTDRV.EXE and Ctrl-Alt-Del, 53
  startup steps, 92
  support, 96-104
Windows 3.0
  DOSX.EXE, 560-561
  FCB functions, 14
  real mode, 122-123
  UMB allocations, 43
Windows 3.1
  FastDisk, 57
  LastV86Page, 42
Windows SETUP XMSMMGR.EXE, 19
WINOA286.MOD, 546-547, 566
  standard mode, 540
  structural representation of EMBs, 540
WriteDisk function, 687, 700
WriteStorage function, 707
WriteTrack function, 694, 700
WSWAP task switcher, 541, 547-548

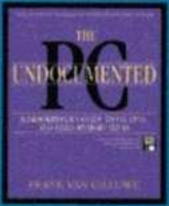
XA20 program
  MAIN.C listing, 404-407
  XA20.C listing, 403-404
XA20.C listing, 403-404
XCOPY utility, 585
XMS functions
  01h (Allocate HMA), 542
  02h (Free HMA), 542
  08h (Query Free Extended Memory), 542-543
  09h (Allocate Extended Memory Block), 543-544
  0Ah (Free Extended Memory Block), 544-545
  0Bh (Move Data to or from Extended Memory Block), 545
  0Ch (Lock Extended Memory Block), 546
  0Dh (Unlock Extended Memory Block), 546-547
  0Eh (Get Information about Extended Memory Block), 547, 566
  0Fh (Reallocate Extended Memory Block), 548, 576, 582
  10h (Allocate Upper Memory Block), 549
  11h (Free Upper Memory Block), 549
XMS.ASM listing, 513-519
XMSCHAIN.C listing, 507-509
XMSFILE program, 459-471
  DOS kernel sidedoor entrance, 489-493
  DOS reentrancy, 488-489
  initialization, 472-478
  MAIN.C listing, 472-477
  makefile listing, 503-504
  memory from SMARTDrive, 478-503
  SDINIT.C listing, 502-503
  SMARTDRV.C listing, 491-499
  task switching, 466-472
  TRANSFER.C (XMS version), 460-466
XMSFILE.C listing, 714-717
XMSFUNC.C listing, 426-427
XMSFUNC.H listing, 425-426
XMSMMGR.EXE, 19
XMSWATCH program, 423-437, 558
  /INSTALL option, 519, 599
  API.C listing, 526-528
  configuring SMARTDRV.SYS, 567-573
  CONSULT.C listing, 528-530
  extended memory blocks, 427-437
  fragmented extended memory and DOSSWAP, 573-577
  INSTALL.C listing, 523-526
  MAIN.C listing, 551-555
  MAKEFILE.C listing, 555-556
  MEMOWNER.C listing, 511-512
  MONITOR.C listing, 520-523
  reporting on DOS 5 configuration, 557
  reporting on HMA usage, 423-425
  SDCONFIG.C listing, 567-573

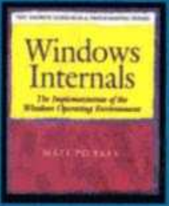
    serious flaws with EMB reallocation, 577-582
    WATCHED.C listing, 530-538
    Windows 3.0 standard mode, 559-577
    Windows enhanced mode, 582-584
    XMS.ASM listing, 513-519
    XMSCHAIN.C listing, 507-509
XMSWATCH.H listing, 427-429

DOS PROGRAMMING

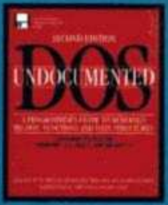
THE ANDREW SCHULMAN PROGRAMMING SERIES

Undocumented Windows


The PC Undocumented


Windows Internals

Windows++


Undocumented DOS

ADDISON-WESLEY
PUBLISHING COMPANY

DOS INTERNALS

GEOFF CHAPPELL

A DOS Anatomy for Serious Programmers

DOS Internals is the most detailed description ever written of how DOS memory and disk management work. By understanding the inner workings of DOS 5 and 6 and the changes made from earlier versions, ambitious programmers will be better able to write programs that take full advantage of DOS services.

DOS Internals is the definitive guide for showing advanced DOS programmers how to exploit the full potential of DOS, and is undoubtedly the most well-researched, meticulous coverage of DOS memory and disk management ever published. This book is divided into four parts: Memory Management, Development Tools, Extended Memory Management, and Disk Management. You'll understand–and learn how to exploit in your programs–key operating system activities, including:

- DOS/Windows interaction
- the DOS boot sequence (including DOS 6 multi-config)
- the DOS/DoubleSpace interface
- extended memory sources, access, and management
- device drivers
- and error codes.

DOS Internals is not a reworking of material found in other DOS books. Current through DOS Version 6, it is based entirely on fresh research and supports the new generation of hardware and software that will be the foundation of personal computing for the rest of this decade.

The accompanying disk comes with complete source code in C and assembly language, including two ready-to-use libraries for turning almost any C program into a DOS TSR or device driver.

DOS and Windows programmers will find in this ultimate DOS reference precious and applicable information available nowhere else.

Geoff Chappell is a doctoral candidate in neural network theory at King's College London. He has written for *Dr. Dobb's Journal* and has conducted extensive professional research on the DOS and Windows operating systems.

53995

9 780201 608359

ISBN 0-201-60835-9

$39.95 US
$51.95 CANADA